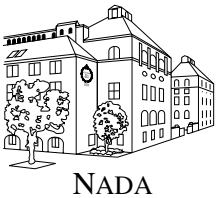


NADA
Numerisk analys och datalogi
Kungl Tekniska Högskolan
100 44 STOCKHOLM

Department of Numerical Analysis
and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, SWEDEN

Annoteringsverktyg för korpusarbete med Granska som grund

Examensarbete

Andreas Aarflot

TRITA-NA-Eyynn

Sammanfattning

Inom språkforskningen organiseras stora textmängder i s.k. *korpusar*. En stor svensk korpus är för närvarande under uppbyggnad på Nada, institutionen för numerisk analys och datalogi, en gemensam institution för KTH, Kungliga Tekniska högskolan, och SU, Stockholms universitet, i Stockholm. Man har hittills samlat ihop över 60 miljoner ord. Korpusen är annoterad – varje ord är försett med tilläggsinformation som ordklass, grundform och meningstillhörighet. Att lägga till sådan information kallas att tagga texten.

För att rätta och underhålla korpus behöver man kraftfulla verktyg. Det här examensarbetet, som är ett större arbete uppdelat på två personer, går ut på att konstruera en prototyp till ett sådant verktyg. Verktyget ska bestå av ett användargränssnitt och en logisk del. Gränssnittsdelen har Helena Ihrfors haft hand om och den logiska delen har jag utfört. I kärnan på den logiska delen finns grammatikgranskningsprogrammet *Granska*, som är utvecklat på Nada. I denna rapport behandlas den logiska delen samt den gemensamma delen med överföringen mellan programdelarna.

För att sökning ska ske effektivt görs förindexering av korpus. Jag har undersökt två indexeringsssätt; på taggbigram och på ord (token). De visar sig båda ge rimliga söktider. Taggbigram ger ofta ett stort antal träffar, speciellt vid sökning på vanligt förekommande taggar. I en fullt utbyggd version av verktyget föreslår jag att även indexering på taggtrigram införs. Taggtrigram ger då en mer precis sökning i korpus och därmed kortare söktider.

Verktygets användardel har skrivits i Java och logikdelen i C++ (Granska är implementerat i C++). Programdelarna ligger separat och kommunicerar med varandra över nätverket. Vi har i den gemensamma delen tittat på olika tekniker för kommunikation mellan de olika programspråken samt lösningar av nätverksuppkopplingen. Valet blev en socket/XML-uppkoppling.

För att underlätta annoteringsarbetet för användaren (en språkforskare) har jag implementerat nya funktioner i Granska, som gruppering av rättelseförslag, slumpning och begränsning.

Slutligen har jag provkört prototypen på *Stockholm-Umeå Corpus*, SUC, med lovande resultat. Det är självklart mycket arbete kvar fram till ett färdigt verktyg med bl.a. ytterligare optimering av vissa programdelar, fullständig felhantering och en grundlig testning av alla funktioner, men alla grundläggande funktioner är implementerade.

Construction of an Annotating Tool for Corpus Correction and Maintenance

Abstract

Part-of-speech text tagging is often used in language technology research. Every word and punctuation mark in a text are assigned extra information such as word class, lemma and sentence belonging. Large quantities of text are collected in a corpus. At the Department of Numerical Analysis and Computer Science (NADA), a joint department of The Royal Institute of Technology (KTH) and Stockholm University (SU) in Stockholm, a large Swedish corpus is now under construction consisting of sixty million words thus far.

Today there is a need for powerful tools that can handle corrections and maintenance of corpuses. This Master's thesis attempts to construct a prototype for such a tool. The work is divided into two major parts: a graphical user interface developed by Helena Ihrfors and a logical part, based on the grammar checking program *Granska*, for which I am responsible. A smaller common part is the network communication between the two components.

In order to make search and replace in the corpus more efficient, pre-processing and optimizing are used. In this thesis, I have looked at two different ways of optimizing; based on tag bigrams and on words (tokens). Both of them give a reasonable short search time. To enhance search speed further, I suggest that a tag trigram optimization will be used in future versions of the tool.

The graphical part of the tool will be implemented in Java, and the logical part will be implemented in C++ (Granska is implemented in C++). Together we have looked at different techniques for the data transfer between the two different programming languages and solutions for the network communication. A socket/XML-connection turned out as the best solution.

A couple of new functions have been implemented within Granska to reduce the user's (a language researcher) effort in the interactive work with corpus. One such function is the grouping of presented corrections (*group*). Others are *randomize* and *limit* functions.

Finally, I have tested the prototype against the Stockholm-Umeå Corpus, SUC, with promising results. A great deal of work still remains for the tool, regarding further optimization of certain functions, complete error handling and exhaustive testing of all functions.

Förord

Denna rapport är dokumentation av ett examensarbete i datalogi vid Stockholms universitet. Examensarbetet omfattar 20 högskolepoäng och har utförts vid institutionen för numerisk analys och datalogi, Nada, KTH och SU, i Stockholm. Arbetet ingår i ett större examensarbete för två personer, där Helena Ihrfors har utfört gränssnittsdelen och jag har utfört den logiska delen. En gemensam mindre del utgör nätverksöverföringen.

Jag vill tacka följande personer:
Prof. Stefan Arnborg, min examinator på Nada.
Johnny Bigert, min handledare på Nada och samtidigt min uppdragsgivare inom språkgruppen på Nada.
Prof. Viggo Kann, min handledare på Nada.
Jonas Sjöbergh, Nada, för hjälp och svar på alla mina frågor.
Björn Eiderbäck, Nada, för råd gällande nätverksdelen.

Andreas Aarflot, augusti 2003.

Innehåll

Kapitel 1

Inledning

I detta kapitel beskrivs bakgrunden och syftet med examensarbetet, samt de i arbetet ingående uppgifterna. Dessutom ges en översikt av rapportens struktur och de olika områden som arbetet har omfattat.

1.1 Bakgrund

För att träna språkgranskningsverktyg för naturligt språk krävs att man har stora mängder text. Till texten vill man ha tilläggsinformation som beskriver ordens grammatiska egenskaper. Att lägga till sådan information, kallas att annotera texten och resultatet blir en korpus – en stor annoterad textmängd. Ett exempel på korpus är *Stockholm-Umeå Corpus* (SUC) [25], som innehåller en miljon ord. Språkgruppen på Nada, institutionen för numerisk analys och datalogi, KTH och SU, i Stockholm har samlat ihop en stor mängd text, över 60 miljoner ord. Från detta ska man bygga en stor svensk korpus.

För att förenkla arbetet med annotering, rättning och sökning i korpus behövs kraftfulla verktyg. Det här examensarbetet går ut på att utforma och bygga en prototyp till ett sådant verktyg. Verktyget ska innehålla ett användargränssnitt implementerat i Java och en logisk del implementerad i C++. Kärnan i den logiska delen ska utgöras av språkgranskningsprogrammet *Granska*, som utvecklats på Nada. Verktyget ska också ha tillgång till en eller flera korpusar i en lokal katalog.

Granska tar vanlig text som indata och producerar med hjälp av ca 250 matchningsregler rättningsförslag som matas ut. I det nya verktyget ska Granska användas på ett annat sätt. En användare (språkforskare) har på sin dator ett GUI (Graphical User Interface) där han/hon anger *en* matchningsregel och anger vilka texter i korpus som ska bearbetas. Detta skickas över nätverket till en server där Granska analyserar texten med hjälp av den angivna matchningsregeln. Rättningsförslagen skickas sedan tillbaka till GUI:t där användaren i ett svarsfönster manuellt markerar de av honom/henne godkända rättningsförslagen.

Mitt examensarbete är den ena delen av ett större examensarbete. Detta består av en gränssnittsdel som Helena Ihrfors har utfört och en logikdel som jag har ägnat

mig åt. En gemensam överlappande del är dataöverföringen mellan gränssnittsdelen och logikdelen.

1.2 Uppgiften

I detta avsnitt följer en sammanfattning av uppgiftens ingående beståndsdelar. Den är uppdelad på en gemensam del och den logiska delen, min del.

1.2.1 Gemensamma delen

- Gränssnittet ska implementeras i Java och logiken i C++ (eftersom Granska är skrivet i C++). Hur ska kommunikationen mellan programspråken, samt nätverksöverföringen, fungera?

- Vi ska gemensamt konstruera ett system för hela verktyget, bl.a. med tanke på var olika data ska lagras och vilket protokoll som ska gälla mellan våra respektive delar.

1.2.2 Logikdelen, min del

- För att programmet ska söka effektivt i korpus ska indexering av texterna göras. Indexeringen ska ske både på ord och på taggbigram, men också implementeras så att andra optimeringssätt är möjliga att lägga till. Sökregistren kommer att bli mycket stora. Jag måste alltså hitta lämpliga lagringsstrukturer för dessa.

- En katalogstruktur ska konstrueras för korpus- och indexfiler.

- Granska som tar vanlig text som indata ska modifieras så att redan taggad text matas in istället, dvs. sök- och ersättlogiken med Granska som grund måste implementeras.

- De nya funktionerna *gruppering*, *slumpning* och *begränsning* ska implementeras. Antalet rättningsförslag som skickas till användargränssnittet kan bli stort. Genom att gruppera (sortera) materialet får användaren bättre överblick och kan därmed kontrollera rättningsförslagen på ett effektivare sätt. Slumpning innebär att en slumpfunktion styr vilka rättningsförslag som presenteras. Begränsning sätter ett tak för antalet rättningsförslag.

- I min del ingår också att sätta ihop delarna till en fungerande prototyp.

- Slutligen ska verktyget provköras på stor korpus.

1.3 Rapportens struktur

I kapitel 2 presenterar jag Granska, taggning av text och hur korpusar är uppbyggda. I kapitel 3 redogör jag för hur vi tillsammans har analyserat och löst nätverksöverföringen, programspråkskopplingen, protokollet mellan delarna och systemarkitekturen för hela verktyget. I kapitel 4 beskriver jag mitt arbete med indexering av korpusfilerna, i kapitel 5 implementationen av sök- och ersättlogiken, i kapitel 6 de nya funktioner som lagts till i Granska och i kapitel 7 provkörning av prototypen samt tidtagning. Slutligen i kapitel 8 följer slutsatser, diskussion och rekommendationer för fortsatt arbete med verktyget.

Kapitel 2

Granska, taggning och korpus

I detta kapitel redogör jag för språkgranskningsprogrammet Granska och dess regelspråk. Vidare presenterar jag taggning av text och beskriver hur stora mängder text insamlas i korpusar.

2.1 Vad är Granska?

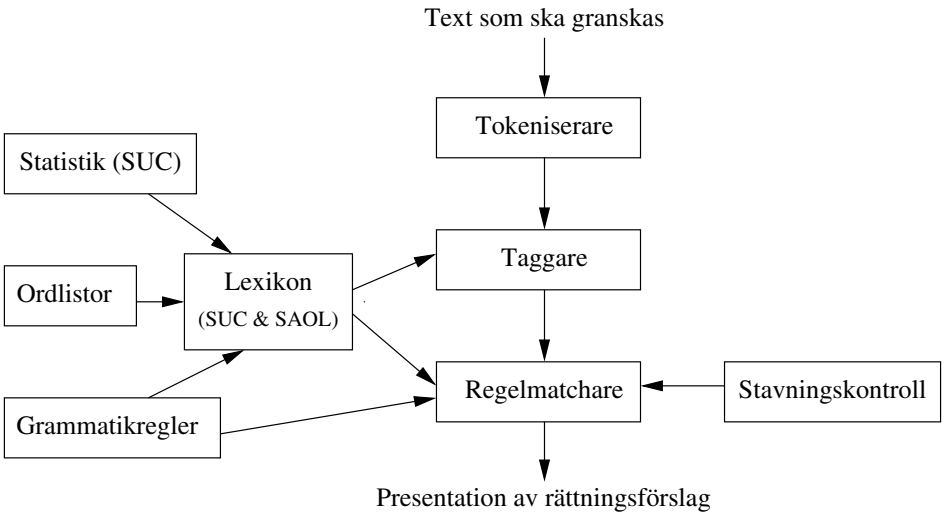
Granska är ett experimentellt språkgranskningsprogram som utvecklats inom projektet *Integrerade språkverktyg för skrivande och dokumenthantering* på Nada, KTH, i Stockholm. Projektet har som mål att utveckla datorfunktioner för språkgranskning på svenska, att införa dessa i en interaktiv miljö för datorstött skrivande och att testa dem empiriskt med användare [12].

Granska tar vanlig text som indata. En granskningsfunktion analyserar språket i texten och markerar de problem den hittar. Till exempel kan programmet hitta stavfel, stilavvikelse och grammatikfel. För att hitta dessa fel använder sig Granska av en uppsättning regler, ca 250 stycken, som beskriver vanliga skrivfel. Reglerna baseras på i huvudsak Svenska språknämndens skrivregler [6], SAOL [5], Myndigheternas skrivregler [19] och Svarta listan [27], [11].

Felaktigheter i texten hittas 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*. Granskasystemet innehåller även en rättstavningsmodul, *Stava* [13], [24].

För att matchning ska kunna göras måste varje ord i texten förses med ordklassinformation, en så kallad *tagg*. Först när texten taggats kan Granska leta efter fel genom att matcha regler mot taggade ordsekvenser [20]. En överblick över Granskasystemet visas i figur 2.1.

Först görs en igenkänning av alla textens ord i *tokeniseraren*. I nästa steg tilldelas orden information om ordklasser och morfologiska särdrag med hjälp av *taggaren*. Taggaren slår upp ordet i ett lexikon och räknar ut ordklasstilldelningen med hjälp av statistik och det sammanhang ordet står i. Den taggade texten skickas sedan till *regelmatcharen* som matchar den mot de förväntade grammatiska avvikelser som



Figur 2.1. *Granskasystemets uppbyggnad [9].*

definierats i granskningsreglerna. Felbeskrivningar och rättningsförslag presenteras därefter för användaren.

En svårighet vid taggning är att många ord i en text kan tolkas på olika sätt beroende på sammanhanget. Ett exempel är ordet *för*, som kan vara både preposition, substantiv och verb. Granska använder sig av *Stockholm-Umeå Corpus* (SUC), en handtaggad korpus, som träningstext och hämtar där statistik på alla förekommande följder av två och tre taggar. Med hjälp av en andra ordningens Markovmodell, som tittar två steg bakåt för varje ny tagg som ska bestämmas, gör Granska kvalificerade ”gissningar” på mest sannolika taggning. Resultaten från två- och tretaggsföljder viktas sedan ihop och på detta sätt disambigueras orden.

Taggaren i Granska är i själva verket en hybrid mellan denna stokastiska taggare och en regelbaserad taggare. I komplicerade fall där den stokastiska taggaren kan ha fel används taggningsregler hämtade ur SAOL [5] för att hitta den korrekta taggningen [10].

Valet av antalet taggar, n, i statistikjämförelserna baserar sig på att om n är litet, $n = 1, 2$, får vi säker men för lite information. Med för stort n, $n \geq 4$, får vi för gles och därför osäker information, men för $n = 3$, dvs. *trigram*, får vi den bästa informationen [7].

En annan svårighet vid taggning är att svenska språket innehåller så många ord och att nya ord konstrueras kontinuerligt genom sammansättningar. Genom att kombinera informationen om vilka taggsekvenser som är vanliga, med en morfologisk analys av ordet, kan rätt tagg gissas med ganska stor precision. Om ordet *puddingarna* är okänt, gissar taggaren sannolikt att det är ett substantiv i plural eftersom det är den statistiskt sett vanligaste taggen för ord som slutar på *-arna*. Sammantaget taggar Granska korrekt med 92 % säkerhet för okända ord och upp till 97 % säkerhet för samtliga ord [10].

```
Rule 1

mitt_kongruensfel@kong
{
    X(wordcl = dt),
    Y(wordcl = jj),
    Z(wordcl = nn & gender != X.gender)
-->
    corr(X.form(gender := Z.gender))
    action(scrutinizing)
}
```

Figur 2.2. *Fiktiv matchningsregel i Granska.*

Taggningsmodulen i Granska processar med en hastighet som är högre än 20 000 ord per sekund på en SUN Sparc station Ultra 5. Hela systemet processar ungefär 3 500 ord i sekunden, taggningen inräknad [9]. Systemet är implementerat i C++ under UNIX (finns även på Win32), för höga prestanda och portabilitet, och kan testas i ett enkelt webbgränssnitt. Adressen är http://www.nada.kth.se/theory/projects/granska/.

2.2 Granskas regelspråk

Regelspråket har en notation som hämtat inspiration från objektorienterade programmeringsspråk som C++ och Java [3]. En regel har två delar separerade med en pil. Den första delen, kallad *vänsterledet*, anger vad som ska matchas i indata. Den andra delen, *högerledet*, anger vad som ska utföras på det matchade indatat.

En matchningsregel i Granska som hittar felet *en runt bord* och ersätter med rättelsen *ett runt bord* kan implementeras enligt figur 2.2.

X symboliserar det första ordet (token), Y andra ordet och Z det tredje ordet för en följd av tre ord i en mening. Nyckelordet *wordcl* betyder särdragsklass. Det finns ett flertal särdragsklasser, varje särdragsklass kan ha ett antal särdragsvärden (se avsnitt 2.3).

I regeln i figur 2.2 har vi $X(wordcl = dt)$ som betyder att ordet som matchar ska vara en determinerare, *en* eller *ett*. $Y(wordcl = jj)$ betyder att ordet ska vara ett adjektiv. $Z(wordclass = nn \ \& \ gender \ != X.gender)$ betyder att ordet ska vara ett nomen (dvs. substantiv), där gender, dvs. genus är skilt ifrån ordet X:s genus.

I högerledet anger $corr(X.form(gender := Z.gender))$ att ordet X:s genus ska korrigeras och sättas till lika med ordet Z:s genus. *corr* är alltså ett kommando för att korrigera. Andra kommandon är t.ex. *mark*, markera, *info*, kommentarer till användaren och $action(scrutinizing)$ som anger att granskning ska utföras.

Granska kan också matcha på ordalydelser. Om det i en regel t.ex. står *word= "katt"*, så matchar Granska på ordet *katt*, dvs. bara när ordet förekommer i en mening så tillämpas regeln. (Se figur 2.3)

```
katt@konkordans
{
    X(wordcl = dt),
    Y(word = "katt")
-->
    mark(Y)
}
```

Figur 2.3. *Matchningsregel med angivet ord.*

Om både taggar och ordalydelse är angivna i sökregeln använder Granska statistik för att välja det ovanligaste alternativet – Är den speciella taggkombinationen vanligare eller ovanligare än det angivna ordet i svensk text? – Genom att välja det minst sannolika alternativet minskas mängden text som behöver granskas och därmed ökar prestanda.

Granska arbetar sig sedan igenom mening för mening i texten, testar varje par av ord från vänster till höger om det matchar mot sökregeln. En utförlig beskrivning av Granskas arbetsätt och språksyntax finns i *Automatisk Språkgranskning av text*, Ola Knutsson 2001 [3].

2.3 Mer om taggning av text

I *part-of-speech* (POS)[1] taggning av text tilldelas varje ord (token) och skiljetecken en morfosyntaktisk tagg. Olika taggningssystem använder sig av olika taggupp-sättningar. En tagg beskriver t.ex. ordklass, numerus och genus för ordet. Antalet taggar i olika system varierar mellan ett dussin och flera hundra. Granskas taggningssystem, som är en vidareutveckling av taggningen i SUC, har 149 olika taggar definierade [10].

I Granska tilldelas varje ord förutom taggen också ett *lemma*, som är ordets grundform. För t.ex. ordet *bilen* är lemmat *bil*. Skiljetecken som punkt, frågetecken och utropstecken ges en speciell tagg, kallad *mad*, som betyder meningsavskiljare. Med hjälp av dessa kan programmet avgöra var meningar börjar och slutar.

Ett exempel på taggning ges i tabell 2.1.

Tabell 2.1. *Taggning av text (lemman ej angivna) [18].*

text	**taggad text**
Den svarta katten	*Den* (dt.utr.sin.def) *svarta* (jj.pos.utr/neu.sin.def.nom)
satt på det lilla bordet.	*katten* (nn.utr.sin.def.nom) *satt* (vb.prt.akt) *på* (pp)
	det (dt.neu.sin.def) *lilla* (jj.pos.utr/neu.sin.def.nom)
	bordet (nn.neu.sin.def.nom) . (mad)

[1]Part of speech: *ordklass*, taggning baserad på fin uppdelning i ordklasser.

En tagg innehåller alltså en uppräkning av särdragsvärden, tillhörande olika särdragsklasser, som beskriver ordet. För några exempel se tabell 2.2 och 2.3.

Tabell 2.2. *Exempel på särdragsklasser och särdragsvärden [3].*

särdragsklass	**betydelse**	**särdragsvärden**
wordcl	ordklass	dt, jj, nn, pp, vb, ab, pm ...
gender	genus	utr, neu, utr/neu, mas
num	numerus	sin, plu, sin/plu
case	kasus	nom, gen
spec	species (bestämdhet)	ind, def, ind/def
vbf	verbform	prs, prt, inf, sup, imp
cht	teckentyp	mad, mid, pad

Tabell 2.3. *Exempel på särdragsvärden (Femininum saknar särdragvärde) [3].*

särdragsvärde	**betydelse**	**exempel**
dt	determinerare (artikel)	den, det
jj	adjektiv	svart
nn	nomen (substantiv)	katt
pp	preposition	på
vb	verb	sitta
ab	adverb	mycket
pm	egennamn	Lars
utr	utrum	en
neu	neutrum	ett
utr/neu	utrum/neutrum	de
mas	maskulinum	bleke
ind	indefinit (obestämd)	en
def	definit (bestämd)	den
vard	vardagligt ord	jobb
mad	skiljetecken vid meningsslut	.!?

2.4 Korpus

En korpus är en stor textmassa som används som underlag för textkritik, analys, konkordans [2], frekvensordbok eller liknande inom språklig databehandling eller datalingvistik. Den kan vara i form av grammatiskt taggad text eller enkel text. Det latinska ordet corpus betyder kropp [26].

En svensk korpus är *Stockholm-Umeå Corpus*, SUC [25], som innehåller drygt en miljon ord fördelat på ca 500 texter med 2 000 ord vardera. SUC innehåller texter av olika typer från skönlitteratur, facklitteratur, tidningstext, myndighetstexter m.m., samtliga tryckta på 1990-talet. Materialet är fritt för användning inom icke-kommersiell forskning, det finns också tillgängligt på CD komplett med ordlistor och frekvenstabeller.

Parole är ett EU-projekt, avslutat 1997, som var inriktat på att bygga upp ett europeiskt nätverk av språkliga resurser. Den svenska Parole innehåller ca 19 miljoner ord fördelat på romaner, dagstidningar, tidskrifter och övrigt [22].

Den tyska Parole innehåller 20 miljoner ord uppdelat på fyra domäner; böcker, tidningar, tidskrifter och diverse [23].

En engelskt korpus är BNC, *The British National Corpus* [14], som innehåller 100 miljoner ord hämtade från en mängd olika källor, från både tal- och skriftspråk.

På Nada bygger man en stor svensk korpus. Hittills har man samlat ihop över 60 miljoner ord.

En korpus innehåller alltså ett mycket stort antal ord fördelade på en mängd texter. I det här examensarbetet arbetar jag med korpus i ett format som förslagits av Johnny Bigert och Jonas Sjöbergh inom CrossCheck-projektet på Nada. Formatet har inget officiellt namn, i rapporten benämner jag det helt enkelt *korpusformat* så länge det inte finns risk för någon sammanblandning. I detta format är texterna kodade med XML och uppdelade på följande filer:

Tokens Här anges ordet (*token*) som det står i källtexten. Token är oftast ett enskilt ord men kan också var sammansatt, t.ex. *den 1 juni 2003*.

Tags Här anges ordklassen (ordets *tagg*).

Lemmas Här anges ordets oböjda grundform, s.k. *lemma*.

Taglemma En kombination av ordets tagg och lemma.

Tags inverterat En ordnad uppräkning tagg för tagg. För varje tagg anges var den förekommer i texten.

Lemmas inverterat En ordnad uppräkning lemma för lemma. För varje lemma anges var det förekommer i texten.

Phrase Här anges början och slut på fraser och stycken. För varje stycke anges dess ingående meningar.

[2] En konkordans är en ordbok eller ordlista där varje ord redovisas i sitt sammanhang, t.ex.: ”eller ordlista **där** varje ord”, ”på Gotland **där** klimatet är”, ”Det var **där** polisen fann” [26].

Structure Här anges var meningar börjar och slutar samt av vilken typ meningen är. Typer är t.ex. *paragraph*, *sentence* eller *html*.

Till annoteringsverktyget har jag valt att använda filtyperna *Tokens*, *Taglemma* och *Structure*. Dessa tre ger tillsammans tillräcklig information för indexeringen. För ett exempel på hur en fil kan se ut, se figur 2.4.

Att bygga en korpus med traditionell grammatikgranskning kan innebära mycket manuellt arbete. Om man använder ett dataprogram för att utföra uppgiften har man svårigheten med att konstruera bra matchningsregler, samt att vissa feltyper är svåra att beskriva med regler och lexikon. Dessutom har ett program alltid en liten felprocent vid taggningen. Handtaggade SUC, som Granska använder som statistisk referens, innehåller ca 5 000 felaktiga taggar, en felfrekvens på 0,5 % [10].

Med hjälp av ett annoteringsverktyg, som detta examensarbete går ut på att konstruera, har en språkforskare möjlighet att bearbeta en korpus interaktivt, och kan på så sätt minimera felfrekvensen. Det innebär t.ex. att man kan reducera antalet felaktiga taggar i en träningskorpus som används av ett grammatikgranskningsprogram som Granska. Därmed kan programmets träffsäkerhet vid taggning ökas.

```
<tokens>
    <token id="aa01.wtl.0">Smygrustning</token>
    <token id="aa01.wtl.1">av</token>
    <token id="aa01.wtl.2">raketvapen</token>
    <token id="aa01.wtl.3">Av</token>
    <token id="aa01.wtl.4">MATS</token>
    <token id="aa01.wtl.5">LUNDEGÅRD</token>
    <token id="aa01.wtl.6">DN:s</token>
    <token id="aa01.wtl.7">korrespondent</token>
    <token id="aa01.wtl.8">.</token>
    <token id="aa01.wtl.9">LONDON</token>
    <token id="aa01.wtl.10">.</token>
    <token id="aa01.wtl.11">Avspänningen</token>
    <token id="aa01.wtl.12">mellan</token>
    <token id="aa01.wtl.13">stormaktsblocken</token>
    <token id="aa01.wtl.14">och</token>
    <token id="aa01.wtl.15">nedrustningssträvanden</token>
    <token id="aa01.wtl.16">i</token>
    <token id="aa01.wtl.17">Europa</token>
    <token id="aa01.wtl.18">har</token>
    . . .
    <token id="aa01.wtl.2444">kan</token>
    token id="aa01.wtl.2445">fullständig</token>
    <token id="aa01.wtl.2446">självständighet</token>
    <token id="aa01.wtl.2447">uppnås</token>
    <token id="aa01.wtl.2448">.</token>
</tokens>
```

Figur 2.4. *Utdrag ur tokensfil för texten "aa01" i korpus SUC.*

Kapitel 3

Nätverksöverföring och systemarkitektur

I detta kapitel redogörs för den överlappande gemensamma delen av examensarbetet. Hur vi har arbetat med överföringen över nätverket, kommunikationen mellan programspråken, systemarkitekturen samt andra övergripande frågor.

3.1 Nätverksöverföring och kommunikation

I avsnittet behandlas olika lösningar av kommunikationen mellan de olika programspråken Java och C++ samt överföringen över nätverket.

3.1.1 Java Native Interface

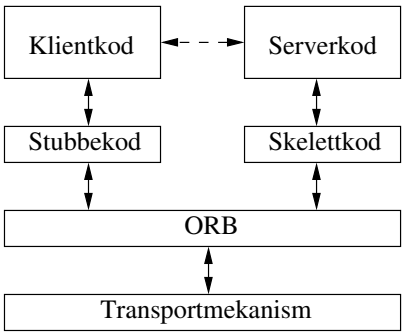
Java Native Interface (JNI) är ett standardgränssnitt för att anropa C, C++ eller assemblerkod från ett Javaprogram. Man kan också innesluta en JVM (*Java Virtual Machine*) i ett C eller C++-program.

Konstruktionsstegen då man från Java vill anropa C++-rutiner är [17]:

1. Skriv Javaprogrammet. Skapa en Javaklass som deklarerar C++-metoden. Den här klassen innehåller deklarationen eller signaturen för C++-metoden, den inkluderar också main-metoden som anropar C++-metoden.
2. Generera en header-fil till C++-metoden med hjälp av *javah* med interface-flaggan *-jni*.
3. Skriv implementationen av C++-metoden.
4. Kompilera header- (.h) och källkodsfilerna (.cpp) till ett gemensamt bibliotek.
5. Kör programmet.

Fördelar: JNI innebär högnivåprogrammering som inte är alltför komplicerad, gränsen mellan programspråken abstraheras bort.

Nackdelar: En blandad C++/Javaapplikation kan inte köras som appletprogram (inte aktuellt i vårt fall). Koden blir kanske inte portabel, den uppbyggda



Figur 3.1. *En CORBA-applikation [2].*

omgivningen blir definitivt inte portabel [16]. JNI är långsamt, vilket kommer att spela stor roll vid överföring av stora datamängder.

3.1.2 CORBA

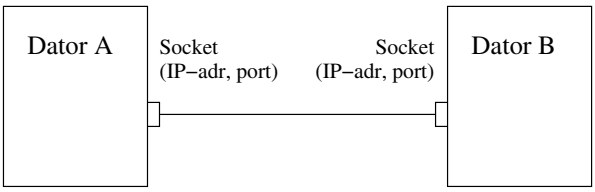
CORBA, *Common Request Broker Architecture* är en arkitektur och infrastruktur som applikationer kan använda för att kommunicera över ett nätverk. CORBA utvecklades av en grupp av över 700 företag med namnet *Object Management Group*, (OMG) i slutet av 1980-talet för att kunna integrera datorer av olika typ och storlek. CORBA är inte en implementation utan en specifikation av den service som systemet erbjuder. Systemet stödjer distribuerade objekt programmerade i en stor variation av programmeringsspråk på olika operativsystem. Med hjälp av CORBA kan metoder exekveras på en annan maskin (jfr. RPC, *Remote Procedure Call*), även ren dataåtkomst är möjlig över nätverket. CORBA fungerar som mellanhand, ett ”lim”, mellan olika programvaror.

En ORB (*Object Request Broker*) sköter om allt det ”magiska” i CORBA. Klientkoden kommunicerar med ett lokalt objekt som innehåller kod för *stubben* (*stub*). Denna kod kommunicerar med ORB:en som utför processer som att identifiera vilket objekt som det ska kommuniceras med. Koden i *skelettet* (*skeleton*) är motsvarigheten på serversidan, där extraheras argument som skickats från ORB:en. ORB:en sköter processer som att identifiera objekt, konvertera data till en ”character”-ström för överföringen samt identifierar servern som innehåller det aktuella avlägsna objektet.

I figur 3.1 visas arkitekturen för en CORBA-baserad applikation.

CORBA används inom e-handelslösningar, webbservrar och är populärt inom finansvärlden för banktjänster, aktiehandel m.m. Ett vanligt användningsområde är servrar som handhar ett stort antal klienter med hög anropsfrekvens, med samtidiga krav på hög driftsäkerhet [15], [21].

Fördelar: CORBA är robust och flexibelt och har en hög abstraktionsnivå. Nätverkskopplingen är transparent, programmeraren behandlar fjärran objekt som om de låg på egen dator. Det är en väletablerad nätverkslösning.



Figur 3.2. *Transport över nätverket med hjälp av socket, en abstraktion av kombinationen IP-adress och portnummer.*

Nackdelar: Aningen hög instegströskel för uppsättande av CORBA-miljön. Prestanda för distribuerade objekt (som CORBA) är sämre än för meddelandesystem (som socket) [2].

3.1.3 Portar och socketar

En kommunikationskanal till en dator som använder TCP/IP-protokollet[1] identifieras av en unik taltupel som består av IP-adressen och portnumret. Denna kombination är en programmeringsabstraktion som kallas *socket* (se figur 3.2). En socket kan alltså sägas vara ena ändpunkten för en tvåvägskommunikation mellan två program som körs i nätverket [2].

Socketar finns av två typer, *serversocketar* som väntar på att klienter ska ansluta sig och *klientsocketar* som ansluter sig till servrar.

En fördel med socket är att den ger möjlighet att på ett smidigt och uniformt sätt kommunicera mellan maskiner av olika typer, den möjliggör vidare programspråksoberoende kommunikation. Det är relativt enkelt att få program skrivna i olika programspråk att kommunicera med varandra.

Varje dator på Internet identifieras med ett fyra-bytes (32 bitars) tal, en s.k. *IP-adress*. Tal är praktiskt för datorer men för människor är det enklare att komma ihåg namn, därför konstruerades DNS, *Domain Name Server* som översätter namn i klartext till nummer. Till exempel översätts `www.nada.kth.se` till `130.237.222.66`.

En *port* är en nätverksabstraktion av en kanal (ledning) in i en dator. Porten identifieras av ett unikt nummer 0–65535. Portnumret används för att hitta rätt process på datorn. Port 0–1023 är reserverade för speciella typer av service, t.ex. 21 för FTP, 25 för mail, 144 för news och 23 för telnet. Port 80 är den normala www-porten. Portnummer mellan 254 och 1023 kan tilldelas till företag för de applikationer de önskar marknadsföra. Portar med nummer över 1023 kan användas för användarapplikationer [30].

Fördelar: Socketar är oberoende av maskintyp och programmeringsspråk, de är relativt enkla att programmera och ger snabb överföring av stora datamängder.

[1] TCP, *Transport Control Protocol* – ett anslutningsbaserat protokoll som erbjuder tillförlitligt datautbyte mellan två noder. IP, *Internet Protocol* – protokollet som används på nätverksnivå av Internet [30].

```
<ENTRY>
    <ENTRYPAIR>
        <NAME>Dodo</NAME>
        <DEFINITION>A dead bird</DEFINITION>
    </ENTRYPAIR>
    <ENTRYPAIR>
        <NAME>Blackbird</NAME>
        <DEFINITION>A thieving bird</DEFINITION>
    </ENTRYPAIR>
    <ENTRYPAIR>
        <NAME>Peacock</NAME>
        <DEFINITION>An attractive bird</DEFINITION>
    </ENTRYPAIR>
</ENTRY>
```

Figur 3.3. *En text uttryckt i XML-språket definierat av DTD-filen i figur 3.4 [2].*

Nackdelar: Socket är en lågnivåabstraktion som kräver mer av programmeraren. Programmeraren får själv utforma ett protokoll för kommunikationen.

3.1.4 Extensible Markup Language – XML

Extensible Markup Language, XML, är egentligen inte ett språk utan en standard för hur man skapar ett ”märkspråk” (markup language) [28]. Den tekniska termen för ett sådant språk är *metaspråk*. XML presenterades 1998 och blev senare en Internetstandard. Uppmärkningen i XML-dokument sker, liksom i HTML-dokument, med hjälp av taggar. I XML är taggarna däremot egendefinierade av utvecklaren. XML-dokumentet har en logisk och hierarkisk struktur där attribut och värden knyts samman. Följande kriterier måste vara uppfyllda för att ett XML-dokument ska anses vara korrekt utformat.

- Dokumentet måste innehålla en unik rotnod som inkapslar hela dokumentet.
- Alla taggar i dokumentet måste motsvaras av en sluttagg, undantaget är om taggen stänger sig själv.
- Inga taggar får överlappa varandra.

Ett exempel på korrekt XML visas i figur 3.3.

XML kan också definiera attribut, dvs. värden associerade till taggar. Syntaxen är:

```
<!ATTLIST elementname attributname attributtype defaultdeclaration>
```

Attributdelen av en sådan rad kan innehålla ett antal fördefinierade värden, till exempel *CDATA* som betyder sträng, *ID* står för en sträng som måste börja med ett alfabetiskt tecken och *ENTITIES* står för multipla namn.

Den valda XML-strukturen måste definieras i en DTD-fil, *Document Type Definition*, innan den kan processas. I DTD-filen anges vilka elementen är, vilka taggar

```
<?xml version = "1.0" encoding ="UTF-8"?>
<!DOCTYPE ENTRY [
<!ELEMENT ENTRY ENTRYPAIR*>
<!ELEMENT ENTRYPAIR (NAME, DEFINITION)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT DEFINITION (#PCDATA)>
<] >
```

Figur 3.4. *En enkel DTD-fil, Document Type Definition, som definierar XML-språket i figur 3.3 [2].*

som associeras med elementen, vilka attribut som kan associeras med ett element och texten som identifierar en tagg.

Ett exempel på en DTD visas i figur 3.4.

Första raden specificerar vilken XML-version och teckenkodning som används, andra raden specificerar att dokumenttypen benämns *ENTRY* och tredje raden säger att en ENTRY ska bestå av en sekvens av noll eller flera *ENTRYPAIR*-element; asterisken betyder noll eller flera element. Definitionen innesluts med klamrar och inleds med *!ELEMENT*. Nästa rad anger att ett ENTRY-par ska bestå av namnet följt av en definition, kommat anger att de två ska konkateneras och alltså ligga på samma hierarkiska nivå. Den femte raden anger att ett *NAME*-element ska bestå av *parsed character data* (PCDATA) (väsentligen icke-taggad text). Den sista raden avslutar definitionen.

Det finns ett antal tecken som kan användas för att beskriva repetition och positionering. Ovan gavs exempel på komma med betydelsen konkatenering och asterisk med betydelsen noll eller flera element. Ett frågetecken symboliserar noll eller en förekomst, ett plustecken står för en eller flera förekomster.

XML har med sin logiska hierarkiska struktur en styrka bl.a. i att kunna överföra data mellan olika format. Data i ordbehandlarformat kan överföras till databasformat som kan överföras till format som läses av en webbläsare etc. [2].

3.1.5 Valet av överföringsmodell

Till en början intresserade vi oss för JNI, och arbetade oss igenom en tutorial på Java Suns hemsida. Det är en aning tilltrasslat i kodandet av de klasser som utgör själva ”limmet”, men ändå överkomligt. Argumenten kan inte utan vidare skickas mellan programspråken, t.ex. är Java-strängar och C++-strängar olika utformade, konvertering måste alltså göras först. Detta kan bli tidsödande när stora datamängder ska skickas. Senare fick vi också uppgifter om att JNI är långsamt. En annan nackdel är att Javaklasser och klasser med själva ”limmet” emellan måste ligga på samma dator som C++-logiken, vilket inte blir flexibelt, det känns renare och enklare om C++-delen är helt separerad från Java-delen.

Framför allt på grund av prestandaproblemen frångick vi JNI och hade då att välja mellan en CORBA- och en Socket/XML-lösning. Båda lösningarna är robusta och flexibla. Vi får bra separation mellan Javadelen, som kan ligga på en dator,

och C++-delen på en annan dator. Utvecklingsarbetet underlättas när varje del kan byggas separat. Den stora skillnaden mellan lösningarna är att CORBA utgörs av högnivåprogrammering och att socket/XML utgörs av lågnivåprogrammering. CORBA anses dessutom något långsammare än socket/XML.

Vi har provat på CORBA i mindre skala i en laboration inom kursen *Internetprogrammering*, men socketlösningen var den som kändes mest näraliggande då vi har provat på tekniken i flera olika kurser och haft bra erfarenheter av den. Det är också så att Granskas utdata är skrivna i XML-format som sedan skickas via en ström till antingen *Standard out*, eller för den webbaserade versionen av Granska, via socket till mottagande webbläsare. Det kändes då naturligt att återanvända dessa redan konstruerade mekanismer i Granska, om än i modifierad form. Att konstruera ett protokoll för XML skulle inte heller bereda några större svårigheter. Vi måste också ta hand om att parsa XML-data när ett meddelande (kommando) eller svar kommer till endera sidan. För det planerade vi att använda redan existerande verktyg, som *DOM-parser* eller *SAX-parser* på C++-sidan. Det tillkommer alltså arbete med att tillägna sig någon av dessa tekniker.

Sammantaget föll till sist valet på socket/XML som den för oss bästa lösningen.

3.2 Systemarkitekturen och överföringsprotokollet

I avsnittet behandlas frågor som rör systemarkitekturen för hela verktyget samt utformandet av överföringsprotokollet mellan gränssnittsdel och logikdel.

3.2.1 Systemarkitekturen

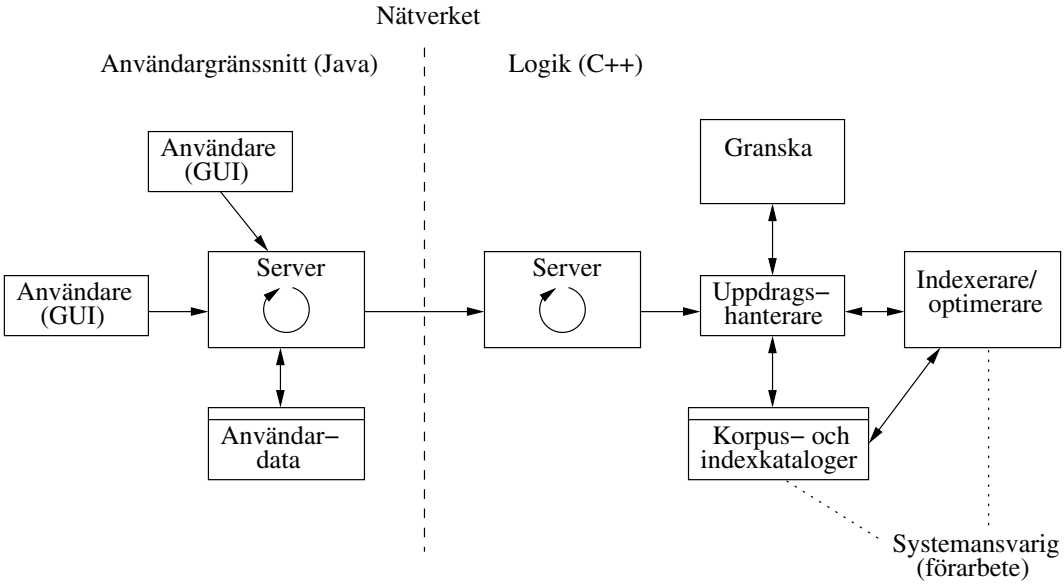
Vi ville ha användar- och logikdelen naturligt avgränsade från varandra där protokollet över ”gränsen” blir så enkelt som möjligt. Användardelen (Java) ska ha en server för att hålla reda på olika användares data, som virtuella korpusar (se nedan i avsnitt 3.2.2), sparade sökfrågor, inloggningsdata och personliga inställningar. Logikdelen (C++) ska ha en server som lyssnar efter uppdrag, tar hand om dessa och skickar tillbaka svar via användarservern till användarna. I logikdelen behövs också en optimerare som dels kan göra förarbete med indexering av texter och dels tar hand om optimering och hämtning av text under en körning av Granska.

Nya korpusar och texter läggs in i logikdelens katalogstruktur av en systemansvarig. Han/hon startar också förarbetet med att indexera de nya korpusarna. Detta görs i en speciell körning av programmet där flaggor markerar indexeringsval, se avsnitt 4.4.1.

Huvudprogrammet sätts igång genom att starta logikdelen. Programmet går igenom ett initieringsskede och ställer sig sedan i vänteläge i en serverloop. Användarservern startas sedan och systemet är redo för att ta emot inloggande användare.

En uppdragshanterare i logikdelen tar hand om inkommande kommandon och administrerar sedan utförandet samt skickar tillbaka data till användardelen.

Systemarkitekturen för hela verktyget avbildas i figur 3.5.



Figur 3.5. *Systemarkitekturen för verktyget.*

3.2.2 Virtuella korpusar

Användaren ska ha möjlighet att själv definiera *virtuella* korpusar. Han/hon väljer ett korpusnamn och anger i ett dialogfönster vilka texter från vilka korpusar som ska ingå. På så sätt kan användaren välja att arbeta mot små eller stora delmängder av korpusar efter behov. En virtuell korpus kan alltså bestå av texter från en eller flera korpusar. Se även figur 4.2 i kapitel 4.

3.2.3 Överföringsprotokollet

Vi ville göra överföringsprotokollet så kompakt som möjligt. De kommandon som visade sig nödvändiga var `RunQuery()` med svaret `Corrections`, `GetTexts()` med svaret `Texts`, `LoadCorpus()` med svaret `Result` samt svaret `Error`. Alla kommandoanrop från Java-sidan har ett förväntat svar. Java-servern står alltså i vänteläge tills svar kommer från C++-sidan, s.k. *synkroniserad* överföring. Då ett kommando terminerar på ett ovanligt sätt returneras istället ett felmeddelande.

Korpusfiler och indexfiler hör självklart hemma på logiksidan, men användardelen måste känna till vilka korpusar och texter som finns. En första idé var att användarservern själv skulle kunna läsa i korpuskatalogen på logiksidan men det skulle bli en alltför inflexibel lösning. En bättre lösning blev nu att vid behov låta användarsidan anropa logiksidan och be om en lista på aktuella korpus/texter. Detta anrop har vi kallat `GetTexts()`. På logiksidan finns då en funktion som tar hand om detta genom att ”scanna av” korpuskatalogen och returnera gällande texter. Nackdelen är att det kan uppstå samordningsproblem mellan delarna.

Om korpusinnehållet på logiksidan ändras, om en text läggs till eller tas bort av den systemansvarige, kan en sparad virtuell korpus då få ett inaktuellt innehåll. För prototypen nöjer vi oss med att detta upptäcks vid körning av en sökfråga. Om en text har tagits bort returneras ett felmeddelande om saknad text. Om en text har lagts till löper allt normalt men sökning sker självklart inte i den tillagda texten så länge den inte finns med i någon virtuell korpus. Däremot, när användaren öppnar en ny virtuell korpus (`New Corpus`), eller editerar en befintlig (`Edit Corpus`) i GUI:t så kommer den nya texten att finnas med som ett valbart alternativ.

I en utbyggd version av verktyget skulle detta kunna implementeras så att varje gång en användare loggar in kontrollerar Java-servern med C++-sidan om någon förändring av korpustexter har gjorts. Om så skett uppdaterar Java-servern sin spegling av korpusinnehållet. Användarna kan sedan underrättas om det ändrade innehållet när det blir nödvändigt.

Att köra en sökfråga har vi kallat `RunQuery()`, rättningsförslagen skickas tillbaka med `Corrections`. Om någonting ovanligt inträffar skickas ett felmeddelande, `Error`. När en sökfråga inte matchar någon mening i någon text eller när en matchningsregel inte hittar några fel används `Result` för att meddela användaren. Samtliga anrops- och svarstyper är kodade i XML-format.

När en sökfråga körs måste logiksidan veta över vilka texter sökning ska göras. Ett sätt är att skicka med data om korpus/textinnehåll varje gång en sökfråga körs, ett annat är att data skickas över med ett speciellt kommando när en användare öppnar en ny korpus i GUI:t. Vi valde det senare alternativet, som vi kallar `LoadCorpus()`, ladda korpus, eftersom en användare troligen kör många gånger i följd mot en och samma korpus, och att körning av en sökfråga (som ändras ofta) ska bli så snabb som möjligt. Nu sparas alltså data om korpus/textinnehåll på logiksidan vid *laddning av korpus*. Vid körning av en sökfråga skickas användarnamnet med som identifikation och logiksidan kan då hämta korresponderande data.

Protokollet mellan användargränssnitt och logik kommer nu att bestå av följande delar (se även figur 3.6).

Funktioner:

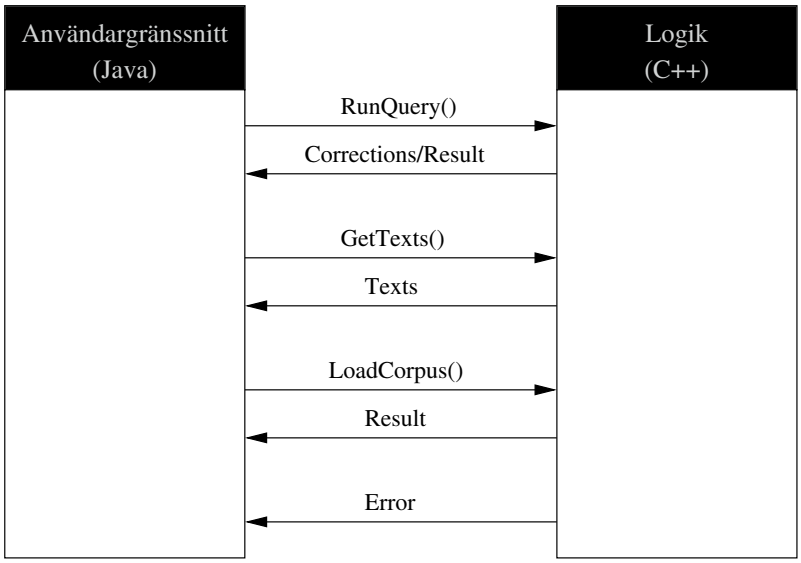
- **RunQuery**, kör sökfråga. *Data*: Sökfrågan, med eller utan grupperingskommando, användarnamnet (som identifierar vilka texter som är aktuella), samt variablerna *randomize* och *limit*.

- **GetTexts**, hämta aktuella texter. *Data*: Användarnamn.

- **LoadCorpus**, ”ladda” korpus. *Data*: Användarnamn, lista på virtuella korpusens namn samt korpus/text-innehållet.

Data:

- **Corrections**, rättelseförslagen från Granska. *Data*: En varierande (kanske mycket stor) mängd rättelseförslag. Kopplad till *RunQuery*, programmet inväntar svaret.



Figur 3.6. *Överföringsprotokollet mellan användargränssnitt och logik.*

- **Texts**, skicka lista på aktuella korpus/texter. *Data*: lista över korpus/texter. Kopplat till *GetTexts*, programmet inväntar svaret.
- **Result**, verifierar att laddningen gick bra. *Data*: Verifikationsmeddelande. Kopplad till *LoadCorpus*, programmet inväntar svaret. Result används också vid *RunQuery* då inga rättelseförslag genererats. *Data*: Information om orsaken.
- **Error**, felmeddelande. *Data*: Beskrivning av felet. Kan ersätta alla svarstyperna ovan.

3.3 Andra övergripande konstruktionsval

I detta avsnitt behandlas övriga frågor som tillhör den gemensamma delen av examensarbetet.

3.3.1 Var ska gruppering utföras?

En gemensam fråga är var *gruppering* ska utföras, i användardelen eller logikdelen. Det är en tänkbar möjlighet att låta användarsidan ha en funktion som givet ett grupperingsalternativ delar upp sökfrågan i flera stycken sökfrågor. Varje sökfråga blir modifierad enligt angiven gruppering. Vid körning anropas då logiksidan flera gånger i följd med de olika sökfrågorna. Rättningsförslagen redovisas sedan efter varandra i svarsfönstret i GUI:t.

En annan möjlighet är att grupperingen sker på logiksidan, att rättelseförslagen innehåller information om grupptillhörighet, men att sortering sker på användarsidan omedelbart innan presentation.

Dessa alternativ är fullt möjliga men båda har nackdelen att systemarkitekturen blir sämre. Det är bättre avgränsat och modulariserat om grupperingen helt och hållet sker på logiksidan. Då är det fortfarande möjligt att byta ut delar av systemet, t.ex. skulle man då kunna koppla in en helt ny användardel mot logikdelen, eller vice versa, om det behovet fanns. Det enda som krävs är då att överföringen sker med socket/XML och att överföringsprotokollet uppfylls.

I avsnitt 6.1 kommer jag att utveckla resonemanget kring gruppering mer i detalj.

3.3.2 Varje ord ska ha ett unikt ID

För att sökning och ersättning i korpus ska vara möjligt har varje ord ett unikt ID, som anger texttillhörighet och ordets löpnummer i texten. När rättelseförslag skickas till användaren måste ordens ID också bifogas. Då kan av användaren godkända rättelser föras in på rätt ställe i korpusen.

Det ur arkitektursynpunkt bästa är att logiksidan ger varje ord sitt ID, som sedan skickas med vid överföringen, men detta innebär också en nackdel. XML är ett ordrikt format i och med sin hierarkiska och tagguppbyggda struktur. Hittills har vårt protokoll tre nivåer, på lägsta nivån ligger hela meningar med insprängda taggar i s.k. *CDATA-block*. Om varje ord dessutom ska förses med ett ID måste vi nästla en nivå djupare, där ord delas upp i ordsträng, tagg och ID. Detta skulle försämra prestanda för överföringen. Mängden data som då ska överföras ökar betydligt. På logiksidan måste också arbete göras med att skicka med varje ords ID hela vägen genom Granska.

Vår lösning är att lägga informationen på meningsnivå. Meningens identitet som den är angiven i korpus, t.ex. `txt4711/14-27`, bifogas meningen och skickas med utmatningen till användarsidan. På användarsidan numreras sedan orden baserat på löpnumret för meningens första ord.

Lösningen innebär ett avsteg från tidigare resonemang om en tydlig uppdelning mellan logik och presentation, men innebär en vinst i prestanda vid överföringen.

Kapitel 4

Indexering av texter

Indexering av texterna i korpus ska implementeras så att programmet vid körning av en sökfråga snabbt kan hämta in det då relevanta textmaterialet för analys i Granska. Den totala lagrade textmassan blir mycket stor, alltså vill man baserat på vad sökfrågan innehåller minimera den delmängd av texten som måste granskas. I detta kapitel undersöker jag lösningar på delproblemet med indexeringen av texterna, förarbetet med indexeringen fram till momentet att utvinna de textbitar som ska analyseras. Följande frågeställningar blir aktuella:

- Att bestämma minsta enhet (block) av text som är lämplig att analysera i Granska.
- Att välja sätt att utvinna optimeringsinformationen ur sökfrågan.
- Att välja lagringsform för indexet.
- Att bestämma hur stor del av textmassan som ska kopplas till varje sökstruktur.
- Att välja sökstruktur vid ordoptimering.
- Att välja sökstruktur vid taggbigramoptimering.

4.1 Konstruktionsval

Detta avsnitt behandlar de grundläggande konstruktionsval som måste göras baserat på bl.a. att optimera verktygets prestanda.

4.1.1 Minsta textblock att analysera

Vid tokenisering i ”original”-Granska delas texten upp i meningar, sedan taggas varje mening separat. Det är inte viktigt att gå över meningsgränser, eftersom korrelationen mellan ord och taggar i intilliggande meningar ofta är väldigt låg. Dessutom är räckvidden på den stokastiska algoritmen bara tre ord, så mycket lite information utbyts över meningsgränserna [10]. Det blir då naturligt att indexeringen som

jag utvecklar ska baseras på meningar (meningsspann) som minsta enhet, dvs. efter matchning hämtas de meningsspann där det matchade ordet eller taggbigrammet ingår. Det betyder att meningarna då kommer att hämtas lösryckta ur sina sammanhang från olika texter, för att sedan analyseras i regelmatcharen i en följd utan någon koppling sinsemellan.

4.1.2 Parsa ut optimeringsinformationen ur sökregeln

I uppgiften var givet att optimeringen skulle göras dels på ord (tokens), dels på taggbigram och att implementeringen skulle göras flexibel så att möjlighet fanns att addera andra optimeringssätt. När en sökregel körs ska på något sätt information utvinnas ur sökregeln om vilka taggbigram och vilket/vilka ord som eventuellt ingår. Det uppstår många kombinationer av taggbigram som är möjliga och att bygga en funktion som utvinner dessa är en komplicerad sak. Nu finns det redan en sådan funktion i Granska som kan vara till hjälp.

Granska har cirka 250 regler och 50 hjälpregler. Varje regel kan matchas i alla positioner (ord) i texten, och det kan också finnas många matchningar mot en och samma regel i samma startposition med olika längd. Regelspråket tillåter operatorerna * (noll eller flera), + (en eller flera) och ? (noll eller en) för tokens, och ; (eller) mellan reglerna, vilket gör att en regel matchar många tänkbara följder av taggar.

Det skulle vara ineffektivt att försöka matcha varje regel i varje position. Därför gör Granska en statistisk optimering där varje regel analyseras i förväg. De tänkbara taggbigrammen för varje position expanderas sedan (se vidare i avsnitt 4.2.2). Med hjälp av relativa frekvenser för taggbigram bestäms vilken av positionerna som är minst sannolik att matcha en mening svensk text. På detta sätt minimeras antalet tillämpningar på regeln när en mening analyseras, vilket ökar prestandan.

Granska upprättar nu två tabeller som också sparas på fil. Den första beskriver, för varje taggbigram, vilka regler som ska tillämpas när just det taggbigrammet förekommer i texten. Den andra tabellen innehåller de ord som förekommer i regeln och beskriver, för varje ord, vilka regler som ska tillämpas när det ordet förekommer i texten [9].

Granska jämför sedan relativa frekvensen för orden med relativa frekvensen för det ovanligaste taggbigrammet. Bara det ovanligaste av dessa lagras i någon av tabellerna. Genom att vara så ”snål” som möjligt minskas alltså mängden text som behöver analyseras.

En naturlig och enkel lösning på problemet med att utvinna optimeringsinformationen blir nu att hämta den från Granskas inre strukturer och använda den i nyskapade sökstrukturer för att söka i korpus.

4.1.3 Lagra informationen på indexfiler

Den totala textmassan kommer att bestå av ett flertal korpusar, var och en med kanske hundratals texter. Denna textmassa kommer att innehålla ett mycket stort antal unika ord. Det finns många böjningsvarianter och det finns ett stort antal sammansatta ord. Ett index över orden blir i nästan samma storleksordning som

källtexten. I en exempeltext med två miljoner ord (`/info/adk01/labb3/korpus`) fanns efter indexering 250 000 unika ordformer.

Utrymmeskravet gör att indexet självklart bara kan lagras på fil (indexfil). Någon form av sökstrukturer konstrueras sedan som, givet en söknyckel, pekar ut var informationen finns på indexfilen.

Hela verktyget kommer att kräva stort diskutrymme. Texterna i korpusarna är skrivna på XML-format där formatet i sig ökar storleken betydligt. XML kan sägas vara ett ordrikt (”verbose”) format om än bra strukturerat och därmed praktiskt. Till det kommer indexfilerna som också är mycket utrymmeskrävande.

4.1.4 Indexera över text, korpus eller totala textmassan

Det lagrade textmaterialet är tänkt att bestå av ett flertal korpusar, var och en innehållande hundratals texter. Frågan är hur mycket textmassa som ska kopplas till en sökstruktur. Ska varje text ha sin instans av sökstrukturen samt indexfiler, eller ska varje korpus ha en egen, eller ska hela textmassan kopplas till en enda, eller finns det något ytterligare alternativ?

Den viktigaste aspekten är självklart att få en snabb sökning, men det är också viktigt att indexeringen i förarbetet blir praktisk och flexibel.

Två saker har stor påverkan på söktiden; totala storleken på antalet träffar (matchningar) och hur många sökningar som görs, dvs. antalet sökstrukturer programmet ska anropa. Om en matchningsregel är allmänt formulerad kan man få ca ett par tusen träffar (meningsspann som ska hämtas in för analys), vilket tar betydligt längre tid att behandla jämfört med kanske hundra stycken träffar för en mer preciserad matchningsregel. Matchningsregelns utformning påverkas bara av användaren, och sökningen kommer då helt enkelt att ta mer eller mindre tid.

En sökstrukturs räckvidd (”scope”) bör ligga nära det omfång en användare mest troligt arbetar inom. Om en sökstruktur har för liten täckning, måste många sökstrukturer processas igenom, vilket tar längre tid, å andra sidan; om en sökstruktur täcker för stort område måste många inaktuella matchningar tas bort, vilket också tar tid.

Som verktyget är tänkt kan användaren välja att arbeta dels mot en hel korpus eller mot en egendefinierad virtuell korpus genom att ange exakt vilka texter som ska ingå. Texterna kan vara från en eller flera korpusar. Användaren kan också givetvis välja att arbeta mot samtliga korpusar.

Den ena ytterligheten är nu att ha en instans av sökstrukturen för varje text. Det innebär hundratals sökstrukturer och indexfiler, dvs. en mycket stor administration. En sökning sker då över varje ingående text. Om användaren arbetar mot ett flertal texter blir söktiden för att söka i många strukturer mångdubbelt längre än att söka i endast en struktur. Endast om användaren arbetar mot enskilda texter (vilket inte verkar troligt) blir detta alternativ bäst.

Den andra ytterligheten är att ha en sökstruktur för alla ingående korpusar. Om en användare arbetar mot alla korpusar skulle det ge en snabb sökning, men troligtvis arbetar en användare mot endast en korpus i taget och då skulle man få en mängd

inaktuella matchningar som måste sorteras ut. Dessutom ska en systemadministratör kunna lägga in nya texter, eller nya korpusar, och då måste i så fall hela textmassan indexeras om varje gång, vilket kan ta 20–30 minuter eller mer.

Jag har valt att ha en sökstruktur för varje korpus, eftersom det verkar mest troligt att en användare kommer att arbeta mot en korpus i taget. Det blir då det snabbaste alternativet. Det blir också praktiskt, då man kan ha ett par färdigindexerade korpusar som inte behöver röras. Vill man lägga till nya texter eller experimentera med specialkonstruerade texter kan dessa adderas i en temporär korpus för sig. Då räcker det att göra omindexering enbart över denna nya korpus. Ett tänkbart scenario är att en användare vill trimma sökfrågorna, och då ha en temporär korpus med specialkonstruerade texter att testa på, samtidigt som systemet fungerar normalt med de andra korpusarna.

En nackdel med den här lösningen är att man får längre söktid vid arbete mot flera korpusar samtidigt, men den försämringen får man acceptera med tanke på högre prestanda i normalfallet.

Någon ytterligare variant med en kombination av ovanstående alternativ känns långsökt, den skulle bli både opraktisk och svårimplementerad.

4.2 Sökstrukturer

I detta avsnitt undersöker jag lämpliga sökstrukturer för ord och taggbigram samt beskriver den expansion av taggpar som Granska gör baserat på informationen i sökfrågan.

4.2.1 Sökstruktur för ord

Enligt ovan kommer indexfilerna att ligga på disk. En sökstruktur ska peka ut positioner i filen. Totala sökrymden blir mycket stor. I Svenska finns det 29 bokstäver i alfabetet, dessutom tillkommer specialtecken om man tillåter sökning på dessa. Om vi för enkelhets skull antar att ett ord är högst 20 tecken långt [1], så finns det alltså 29^{20} möjliga kombinationer, ett mycket stort antal. Då har vi ändå bortsett från att det i svenska kan förekomma mycket långa sammansatta ord.

Tänkbara sökstrukturer:

- sorterad vektor
- binärt sökträd
- trie (träd där varje nivå motsvarar en bokstav i ordet)
- hashtabell
- latmannahashning

En sorterad vektor som lagringsstruktur är tänkbar, om tabellen innehåller n ord blir söktiden blir $log(n)$, det är ganska snabbt; för t.ex. 100 000 ord krävs högst

16 jämförelser. Det är risk för att vektorn blir för stor för att få plats i internminnet, speciellt med tanke på om det senare läggs till nya korpusar. För ett binärt sökträd är nackdelen att pekarna tar stor plats, söktiden blir här också $log(n)$. För en trie blir det också väldigt många pekare, den är dessutom svår att implementera effektivt [29]. Ett alternativ är en hashtabell, som är mycket snabb, men den skulle bli mycket stor och problemet med krockhanteringen måste lösas. Med s.k. *latmannahashning* fås ett bra alternativ; det blir snabbt och tar liten plats i internminnet.

Med tanke på indexets storlek och att det dessutom ska kunna växa ytterligare samt med tanke på prestanda valde jag latmannahashning som det lämpligaste alternativet. De facto använder Granska hashtabeller bl.a. för sitt ordlexikon, där man löst krockhanteringsproblemet på ett smart sätt och där inläsningen sker med fast-filer [10], men det var jag inte medveten om när beslutet togs. I verktyget är det inte heller lika tidskritiskt som i Granska, eftersom det är frågan om att söka på ett eller några ord under en körning. I Granska däremot ska varje ord i texten slås upp och taggas i runtime. Det visar sig senare också att latmannahashning blir snabbt och effektivt.

I latmannahashning hashas bara på de tre första bokstäverna i söknyckeln (ordet). Sedan används binärsökning. Latmannahashning är lämplig för sökning med få diskåtkomster i en stor text när ordindexet inte kan ligga i primärminnet [29].

Idén är att en sökstruktur (tabell) med alla kombinationer av de tre första bokstäverna i ett ord blir rimligt stor och att detaljinformationen sedan ligger lagrad på en mycket stor sorterad indexfil. Ur sökstrukturen hämtas två positionsangivelser till indexfilen, i detta begränsade område i indexfilen binärsöker man sedan efter det fullständiga sökordet och utvinner till sist indexinformationen. Att hasha och slå upp i tabell går i konstant tid. Att binärsöka i indexfilen inom ett smalt område kräver ett litet antal jämförelser. Om vi antar att området består av 100 ord så krävs högst 7 jämförelser. Latmannahashning blir effektivt även för mycket stora indexfiler.

4.2.2 Expansion av taggpar

Antalet unika taggar i Granska är 149 stycken. En tagg innehåller information om ordklass, genus, numerus etc. När en regel är ”generellt” skriven som i figur 4.1 – en determinerare ska följas av ett, inget eller flera adjektiv, som ska följas av ett substantiv – så kommer de möjliga taggbigrammen som matchar regeln att bli ett stort antal.

För regeln gäller att följande kedjor av taggar är möjliga:

```
dt, nn
dt, jj, nn
dt, jj, jj, nn
dt, jj, jj ,jj ,nn
etc.
```

Det beror på att angivelsen '*' på raden efter Y innebär noll eller flera förekomster av taggen *jj*. Den fjärde kedjan ovan matchar t.ex. meningen *Den långa smala vindlande vägen*, som innehåller tre förekomster av adjektiv efter varandra.

Granska har en funktion som expanderar alla dessa möjliga följder av taggar. Sedan beräknas, med hjälp av taggbigramstatistik, vilket par av konsekutiva kolumner som är det minst förekommande i svensk text. Detta kolumnpar använder Granska slutligen i sin regeloptimerare. Min idé är att använda denna information till taggbigramindexeringen.

Om vi för enkelhetens skull nu antar att kolumn 1 och 2 blir utvalda av optimeringen, så har vi redan två taggpar som gäller: *dt/nn* och *dt/jj*. Om vi tittar på regeln vet vi dessutom att för X gäller att $wordcl = dt$, vilket är ganska opreciserat. Detta passar nämligen in på följande 13 taggar:

```
<tag n="4" name="dt.utr/neu.plu.def"/>
<tag n="8" name="dt.utr.sin.ind/def"/>
<tag n="9" name="dt.utr/neu.sin.def"/>
<tag n="18" name="dt.utr/neu.plu.ind/def"/>
<tag n="23" name="dt.utr/neu.plu.ind"/>
<tag n="46" name="dt.utr/neu.sin.ind"/>
<tag n="56" name="dt.utr.sin.ind"/>
<tag n="57" name="dt.utr/neu.sin/plu.ind"/>
<tag n="71" name="dt.neu.sin.def"/>
<tag n="84" name="dt.neu.sin.ind"/>
<tag n="92" name="dt.mas.sin.ind/def"/>
<tag n="93" name="dt.neu.sin.ind/def"/>
<tag n="130" name="dt.utr.sin.def"/>
```

För alla taggarna gäller att $wordcl = dt$. Taggarna har sedan olika variationer av genus, numerus och species.

För Y i regeln gäller att *wordcl=jj*, vilket passar in på 17 olika taggar.

Om vi bara tittar just på taggbigrammet *dt/jj*, finns det nu så mycket som $13 \cdot 17 = 221$ olika taggkombinationer som matchar. För varje av dessa 221 taggbigram, plus de 351 kombinationer som uppstår för bigrammet *dt/nn*, totalt 572 kombinationer, har Granska i en tabell satt en pekare till den aktuella regeln. I min tillämpning ska programmet nu hämta de aktuella meningsspannen för alla dessa

```
kongruensfel@kong
{
    X(wordcl = dt),
    Y(wordcl = jj)*,
    Z(wordcl = nn & gender != X.gender)
-->
    corr(X.form(gender := Z.gender))
    action(scrutinizing)
}
```

Figur 4.1. *"Generellt" skriven matchningsregel som ger många träffar.*

kombinationer. Det visar sig att man får många dubbletter, samma mening pekas ut som resultat av många olika taggkombinationer. Vi vill självklart bara analysera en mening en gång, framför allt inte få samma rättningsförslag i GUI:t flera gånger. Programmet måste alltså antingen ta bort dubbletter i något skede eller fortlöpande markera (pricka av) de meningar som redan hämtats till granskning.

4.2.3 Sökstruktur för taggbigram

Sökstrukturen för taggbigram blir helt enkelt en matris $149 \cdot 149$ stor, dvs. ca 22 000 ”fack”, vilket är en rimlig storlek. I varje fack lagras en positionsangivelse i en stor indexfil över taggbigramförekomster i texterna. En mappning av taggnamnet till taggens id-nummer görs, `nn.neu.sin.def.nom` mappas t.ex. till ’0’, och sedan slås värdet av taggparet upp i tabellen, vilket går i konstant tid. Trots att programmet ofta får läsa ett par hundra värden går det således ändå mycket snabbt.

För att ta bort dubbletterna som uppkommer vid taggbigramsökning använder jag en temporär boolsk tabell över alla meningsspann i en text och prickar av när ett meningsspann är hämtat. Det går nu snabbt att identifiera de meningar där ett visst taggbigram förekommer. Det som tar tid, som det senare visar sig, är att hämta alla dessa meningsspann från de olika filerna och lagra dem i strukturer (objekt) som Granska sedan kan använda vid analys.

4.3 Lagringsstrukturer

Textmassan är lagrad i katalogen `corpus`, som har en underkatalog för varje korpus. Varje korpus har sedan en underkatalog för varje ingående text. I en sådan finns textinnehållet fördelat på följande filer i XML-format:

`structure` – innehåller identitetsnummer för meningars början och slut.

`tokens` – innehåller orden (tokens) som de uppträder i texten.

`taglemma` – innehåller ordens taggar och lemman.

`tags` – innehåller ordens taggar, används inte för närvarande.

`tok-rowindex` – binärfil med radpositioner till tokensfilen för direkt uppslagning.

`taglm-rowindex` – binärfil med radpositioner till taglemmafilen för direkt uppslagning.

Indexfilerna har jag valt att lagra för sig i katalogen *index*, som har en underkatalog för varje korpus. Under varje korpus finns filerna:

`tokens-index` – indexfil för ord.

`tokens-switchindex` – ”mellan”-indexfil för binärsökning som pekar ut positioner i tokens-index.

`tagbg-index` Indexfil för taggbigram.

`word-table` – latmannahashtabellens data på fil.

`tagbg-matrix.txt` – taggbigramsökstrukturens data på fil.

Under katalogen *users* finns en enkel lagring för användares virtuella korpusar. I en utbyggd version av verktyget bör varje användare tilldelas en egen underkatalog för att undvika namnkonflikter. Figur 4.2 visar ett exempel på XML-lagringen av en virtuell korpus skapad av användare *a*. Den innehåller texter från två korpusar, *suc* och *test*.

```
<?xml version="1.0" encoding="UTF-8"?>
<text><textname>suc/ab07</textname>
      <textname>suc/ab08</textname>
      <textname>suc/ab09</textname>
      <textname>suc/ab10</textname>
      <textname>suc/ab11</textname>
      <textname>test/txt4711</textname>
      <textname>test/txt4712</textname>
</text>
```

Figur 4.2. *Lagring av den virtuella korpusen "a.corpus", med dess ingående texter från två olika korpusar.*

4.4 Överväganden vid implementationen av indexeringen

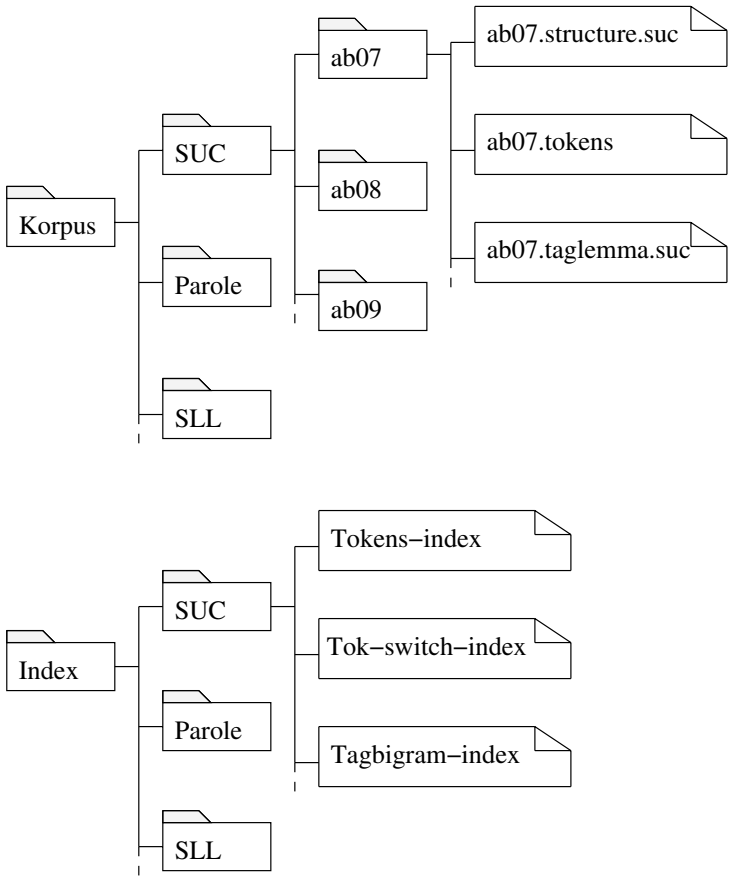
I det här avsnittet behandlar jag implementationen av indexeringen, som består av två delar, den första delen är förarbetet som initieras av en systemansvarig, den andra delen är optimeringen i runtime när en sökfråga körs av en användare.

4.4.1 Indexeringen gör ett förarbete

I ett förarbete ska indexfiler och sökstrukturer byggas. Denna programdel sköts av en systemadministratör. Jag har implementerat två sätt att göra förarbetet på, det första är att indexera samtliga befintliga korpusar och det andra är att indexera endast angivna korpusar (se tabell 4.1).

Tabell 4.1. *Indexeringskommandon.*

kommando	**innebörd**
>scrutinizer -Y -a	optimera allt
>scrutinizer -Y -o *korpusnamn...*	optimera angivna korpusar



Figur 4.3. *Katalogstruktur över korpusfiler och indexfiler. SUC, Parole och SLL är tänkta korpusar. ab07, ab08, ab09 är textnamn. Varje textkatalog innehåller XML-filer med data. Under indexkatalogen har varje korpus sin uppsättning indexfiler.*

Det är självklart praktiskt att kunna indexera om alla korpusar i ett svep, om t.ex. något i programmets implementation ändras. Detta blir med nödvändighet ganska tidskrävande, för programmet måste ju då läsa och parsa igenom samtliga texter, men eftersom det så sällan görs är det acceptabelt. Vanligast blir att indexera endast en korpus med kanske nyinsatta eller ändrade texter. Det kan då göras separat utan att påverka övriga indexfiler. Indexeringstiden för en testkorpus med 191 000 ord fördelade på 80 texter är cirka 5 minuter.

Vid indexering kontrollerar först programmet i katalogen `corpus` (se figur 4.3) om de aktuella korpusarna motsvaras av underkataloger med samma namn, alternativt hämtar alla nu gällande korpusnamn från katalogen. Den systemansvarige måste i den här prototypen också manuellt kontrollera att alla korpusar har en motsvarande underkatalog i katalogen `index`. Här kommer indexeringsprogrammet att lägga de indexfiler som skapas för varje korpus.

För varje korpus skapas en sökenhet, ett objekt som jag kallat för *Table*, i den

sätts tabeller upp för de olika indexeringssätten. För varje korpus hämtas alla dess ingående textnamn. För varje text parsas först de gällande meningsspannen ut från structure-filen, sedan parsas ord för ord ut (eller taggbigram) och läggs till sist i en temporär fil tillsammans med text- och meningstillhörighet på formen:
($ord, text/meningsspann$), t.ex. `abakus text118/14-27`. När alla texter gåtts igenom är samtliga ord (taggbigram) sparade på den temporära filen. Nästa steg är lexikal sortering av tmp-filen med hjälp av `sort()` i C++ (som sätter *'ä'* före *'å'* vilket måste tas hänsyn till i andra delar av programmet). Orden buntas sedan ihop och skrivs till en indexfil.

Eftersom användaren kan köra mot valfritt antal av de texter som ingår i en korpus, måste de ej valda texterna sorteras ut i något skede. Jag har valt att organisera indexfilen så att efter ett ord (sökord) följer först ett textnamn följt av en rad med dess meningsförekomster. På nästa rad står ett nytt textnamn följt av dess meningsförekomster etc. Sökord markeras med *'$'* framför för att kunna skiljas åt från textnamn.

Under körning när information ska hämtas ur indexfilen slås först nyckelordet upp, sedan tittar programmet på textnamnet direkt efter nyckelordet, slår upp i en hashtabell om textnamnet är aktuellt och i så fall hämtar meningsförekomsterna på den raden. Om textnamnet inte är aktuellt går programmet till nästa rad osv. På så sätt sorteras på ett effektivt sätt de inaktuella texternas förekomster bort.

För ett exempel på en indexfil se figur 4.4.

När indexfilen är klar sparas filpositionerna för varje nytt nyckelord i tabellerna i sökstrukturen. I fallet för ord görs här också en mellanindexfil (se avsnitt 4.4.3) för att snabba upp binärsökningen.

Pseudokod för indexeringsförarbetet visas i figur 4.5.

4.4.2 Direktåtkomst med binärfiler

Direktåtkomst innebär förflyttning direkt till en godtycklig plats i filen istället för sekventiell förflyttning från början till slut. Direktåtkomst används ofta med databasfiler. Det sker enklast om filen består av en samling lika stora poster så att varje post representerar relaterade data [4].

Detta används här genom att skapa binärfiler, där post nr i pekar ut positionen för i:te ordet i tokensfilen och analogt för taggbigram och lemma i taglemmafilen. Binärfilerna används sedan när de av indexeringen optimerade meningarna ska hämtas in för analys i Granska. Man vill då inte behöva leta reda på orden i XML-filen för att parsa ut dem ytterligare en gång utan vill kunna slå upp deras positioner direkt. Eftersom binärfilen har poster (structar) av en given storlek kan man enkelt räkna ut var post nummer i finns, hämta en positionsangivelse där och läsa direkt i XML-filen.

```
...
$grund aa08 780-799
aa09 1059-1093 1280-1293
aa11 956-970
aa13 2375-2391 443-457
ab01 229-241
ab02 1790-1803
ac01 2219-2234
ac02 1054-1074 994-1009
ac03 181-195
ac04 1216-1233 1823-1844
ae01 1550-1561
ae03 628-644
af01 2191-2204
ba01 402-418
ba06 1366-1375 811-841
bb06 842-875
bb09 2243-2268
bb10 1642-1678
ca04 1340-1368
cb03 1639-1656
cc03 2015-2055
cd02 357-384
$grund- aa13 2261-2292
$grunda aa10 2094-2112
$grundackord ce01 47-88
$grundad ca02 2264-2280
$grundade aa12 499-512
$grundare af02 1202-1243
$grundat ce03 145-167
...
```

Figur 4.4. *Utdrag ur en indexfil. '$' föregår nytt sökord, aa08, aa09 är textnamn, 780–799 anger meningsspann.*

4.4.3 Uppsnabbning av binärsökning

En nackdel med att binärsöka i indexfilen är att ett vanligt ord följs av många positionsangivelser. Efter vanligt förekommande ord, som t.ex. *att*, *eller*, *och* kan det förekomma hundratals positionsangivelser. Binärsökningen kommer nu oftast att peka på en positionsangivelse och programmet måste då stega fram till nästföljande ord innan det kan avgöra om det sökta ordet är hittat eller står före eller efter i filen, vilket tar onödigt lång tid. För att snabba upp skapar jag en mellanliggande fil som programmet binärsöker i först. Den består av enbart sökordet följt av en positionsangivelse. Här går sökning efter nyckelord snabbt. När ett ord hittats hämtas positionsangivelsen efter ordet. Denna pekar ut positionen för ordet i indexfilen, och där kan indexeringsinformationen då direkt läsas och hämtas.

```
FOR (alla aktuella korpusar)
    // ordoptimering (token)
    FOR (alla texter)
        Läs structurefilen, lagra meningsgränser
        Läs tokensfilen --> gör append i tmpfil med
                                    (ord, textnamn/meningsspann)
    sort(tmpfil) --> gör indexfil
    gör mellanindexfil (för snabb binärsökning)
    sätt in filpositioner i sökstruktur (Latmannahashtabell)

    // taggbigramoptimering
    FOR (alla texter)
        Läs structurefilen, lagra meningsgränser
        Läs taglemmafilen, kombinera parvisa taggar -->
            gör append i tmpfil med (taggbigram, textnamn/meningsspann)
        sort(tmpfil) --> gör indexfil
        sätt in filpositioner i sökstruktur (149*149 matris)

spara tabeller på fil
```

Figur 4.5. *Indexeringsförarbete, pseudokod.*

4.4.4 Implementation av latmannahashning

Latmannahashningen har en tabell med storlek 30 000. Ett sökords tre första bokstäver ger ett hashvärde. Om c_i är bokstav på plats $i = 1, 2, 3$ och om a mappas till 1, b till 2, c till 3 etc. ges hashvärdet av $h = c_1 \cdot 30^2 + c_2 \cdot 30 + c_3$. Alla bokstavskombinationer får då ett unikt hashvärde. Om sökordet är t.ex. xylofon slår man upp på hashvärdet av *xyl*; ur tabellen får man en positionsangivelse i en (mellan-)indexfil med alla förekommande ord. Nästa tänkbara prefix i tabellen blir då blir *xym*. Nu finns troligen inget ord med det prefixet så den platsen i tabellen är tom, men för att slippa stega fram via många tomma platser till nästa ”icketomma”, kopierar programmet alltid in den nästkommande ”icketomma” positionsangivelsen på platsen direkt efter platsen för *xyl*. Denna andra positionsangivelse hämtas. Mellan dessa två positioner i (mellan-)indexfilen, som är sorterad i bokstavsordning, finns ordet *xylofon*, som letas upp med hjälp av binärsökning. När ordet hittats, om det finns, hämtas en ny positionsangivelse (som står direkt efter ordet) till en indexfil över ord samt var de förekommer i källtexten. I denna stora fil står alltså samtliga ord uppräknade, följt av i vilka texter och i vilka meningsspann de förekommer. Till sist returneras en lista över textnamn/meningsspann med alla förekomster av det sökta ordet.

4.4.5 Sökbara tecken

I den nuvarande prototypen har jag implementerat ordsökningen så att endast tecken som tillhör svenska alfabetet är sökbara. Däremot är samtliga i texterna förekom-

mande specialtecken insorterade i indexfilerna (i lexikal ordning enligt `sort` i C++). Nollan i kodningen är reserverad för icke-tecken (tomt). *a* mappas till 1, *b* mappas till 2 etc.

En studie av Granskas ordinarie regeluppsättning visar att vissa specialtecken förekommer, om än sparsamt. Det är tecknen: *"()1?* . I en fullt utbyggd variant av verktyget kan sökning på specialtecken läggas till. Det enda som då behöver ändras är filen med mappningar av tecken till siffra, samt att indexera om samtliga texter. För att inte tabellen ska svälla alltför mycket kan alla de ovanligaste tecknen få en och samma kodning. De lite vanligare tecknen kan ha en unik kodning. *Siffror* $0-9$ kan också ha en och samma kodning. Att söka speciellt på '.' skulle betyda att hämta i princip alla meningar till granskning, ett mycket stort arbete. Tecknen *$*, < och > går inte att söka på eftersom de är "reserverade" tecken i indexfilerna. Understrykningstecknet, *'_'*, kodas som noll, det används som utfyllnad för ord kortare än tre bokstäver i latmannahashningen.

4.4.6 Testprogram för latmannahashning

Ett testprogram som jag först implementerade använder en text, `/info/adk01/labb3/korpus`, bestående av 2 miljoner ord. Givet ett sökord hämtas alla förekommande konkordanser i texten med hjälp av ovan beskriven latmannahashning. Att utföra detta för en lista på 200 slumpvist valda ord, med i snitt 4 konkordanser per ord, tog ca 3 sekunder (konkordanserna skrevs inte på standard out). Hämtningshastigheten blev alltså ca 67 ord/sekund.

Kapitel 5

Integrering med Granska

I detta kapitel tar jag upp arbetet med att integrera mitt program med Granska; t.ex. hur in- och utmatningen till Granska har förändrats och hur funktioner har modifierats.

5.1 Inledning

Granska tar normalt vanlig text som indata. Texten tokeniseras och taggas, meningar byggs och analyseras i regelmatcharen. Till sist matas rättningsförslagen ut. För verktyget gällde nu istället att färdigtaggade meningar direkt skulle matas in i regelmatcharen. För att implementera detta måste jag först ta reda på och förstå hur Granska fungerar internt. Granska är ett stort och komplicerat program. Utan tillgång till dokumentation fanns bara alternativet att studera källkoden (sparsamt kommenterad), provköra programmet, experimentera med inmatning av olika texter och flaggor, göra provutskrifter etc. Jag hade självklart också stor hjälp av handledarna och folk på institutionen. Att spåra förloppen i Granska var en spännande men tidsödande uppgift; många gånger tappade jag bort mig i en labyrint av funktionsanrop, där metoder ofta var nedärvda i arvskedjor.

Granska är kodat av flera upphovsmän under en längre utvecklingsperiod, några finns inte längre kvar på institutionen, så det är svårt att få detaljinformation. Ett tidigt råd från handledarna var att ändra så lite som möjligt i Granskas kod, eftersom konsekvenserna kunde bli svåra att överblicka. De rekommenderade att istället koppla in och ur sig på väl valda ställen i koden med så liten påverkan av originalprogrammet som möjligt. Det visade sig svårt att leva upp till detta, jag blev tvungen att göra en hel del modifieringar eller överlagringar av originalfunktioner.

Granska är mycket effektivt implementerat, men inte så lätt att handskas med då olika programdelar inte alltid är helt frikopplade från varandra. Granska har också globala variabler som är svåra att kontrollera.

5.2 Indata skickas direkt till regelmatchningen

Vid körning av en sökfråga från användaren kommer optimeringen nu att hämta hem de relevanta meningsspannen. De lagras då i en nyskapad vektor med färdigtaggade ord (tokens), där sista ordet (token) i varje mening har en flagga satt för meningsslut. Dessa ska nu till regelmatchningen i Granska för analys. För att inkopplingen till Granska ”från sidan” skulle vara möjlig utan alltför stora ingrepp i originalkoden går jag via en funktion snarlik taggaren i Granska. I Granska sätts här ordens *lemma* och *tagg*, men i min funktion sätts dessa med de värden som hämtats från korpus. Nästa steg är att dela upp texten i meningar. Där har jag också inspirerats av en Granskafunktion, `BuildSentences()`, men min funktion bryter för ny mening enligt data från korpus.

5.3 Granska som server

Granska-implementationen som jag hade tillgång till är gjord för att köras en gång för varje textinmatning, sedan avslutas programmet. I verktyget ska logikdelen däremot fungera som en server. Granska ska vara uppstartad och klar, alla initieringar och inläsningar av lexikon gjorda. Sedan ska sökfrågor kunna köras genom Granska.

Granska finns i en webbversion, som fungerar som en server, men den uppvisar en stor skillnad mot denna tillämpning. I webbversionen matchas texten mot inte *en* matchningsregel utan mot *hela regeluppsättningen* (250 matchningsregler plus 50 hjälpregler). Hela regelfilen kan alltså laddas in vid uppstart av servern och är densamma under programmets hela förlopp. I det här verktyget är dock grundidén att *en* matchningsregel körs åt gången. Vid nästa körning är matchningsregeln antagligen förändrad.

I webbversionen kan alla initieringar göras vid uppstart, inklusive laddning av regelfilen. Detta görs i funktionen `scrutinizer.Load()`. Sedan kan de texter, som tas som indata via webben, analyseras direkt i regelmatcharen med `scrutinize()`, och utmatning skickas tillbaka över webben. I verktyget måste jag nu istället dela upp `scrutinizer.Load()` i två delfunktioner, `scrutinizer.LoadLex()` och `scrutinizer.LoadRule()`. Den första initierar allting som kan ligga konstant under programmets livslängd och den andra laddar en ny sökregel för varje körning.

Jag fick också skriva en ny funktion som tar bort dynamiska strukturer och nollställer efter varje körning.

5.4 Utmatningen från Granska

Utmatningsfunktionen i Granska var förhållandevis enkel att modifiera. Den använder sig av XML-format och bygger upp XML:s hierarkiska struktur med hjälp av ett par funktioner. En funktion konstruerar en ny tagg med angivet namn och värde, en annan avslutar taggen. Dessa funktioner var till stor hjälp och gav alltså möjlighet

att definiera egna taggar. Med hjälp av dessa ändrade jag utmatningsstrukturen så att den motsvarade det överenskomna protokollet mot användargränssnittet.

Andra förändringar av utmatningsfunktionen beskriver jag i kapitel 6.

Kapitel 6

Implementation av nya funktioner

Detta kapitel handlar om tre nya funktioner som har implementerats i Granska; *Gruppering* (*Group*), *Slumpning* (*Randomize*) och *Begränsning* (*Limit*).

6.1 Gruppering

Gruppering avser att svarsdata (rättningsförslagen) från Granska ska kunna organiseras på ett praktiskt sätt för användaren. Verktyget är tänkt att vara interaktivt på så sätt att användaren (en språkforskare) ska kunna godkänna eller förkasta rättelseförslagen som kommer upp i svarsfönstret i GUI:t. Med gruppering får användaren bättre överblick och större möjlighet att organisera sitt arbete.

Bara de godkända rättelserna kommer att användas. I ett första steg (i denna implementering) sparas rättelserna enbart på fil i XML-format. I en utbyggd version av verktyget kommer rättelserna inte heller att införas i originalkorpusen. De kommer istället att isoleras från den underliggande korpusen med hjälp av indirekta referenser via pekare eller via en tabell. På så sätt kan rättelser distribueras fritt utan att den underliggande korpusen behöver följa med. Man slipper då också copyright-problem och att utföra versionskontroller [8].

Gruppering anges i sökregeln med det reserverade ordet *group* skrivet någonstans i högerledet före något av de exekverande kommandona, t.ex. *scrutinize*. Inom parentes anges det som ska grupperas på, som kan vara ett av särdragsvärdena, t.ex. *gender*. Meningen är att rättelseförslagen sedan ska visas i sorterad ordning i svarsfönstret i GUI:t. Den första gruppen anges av grupperingsalternativ 1, t.ex. alla meningar där Z-ordet är utrum. Den andra av grupperingsalternativ 2, t.ex. alla meningar där Z-ordet är neutrum. Den sista gruppen består av de meningar som inte uppfyller något av grupperingsalternativen, dvs. återstoden, där Z-ordet varken är utrum eller neutrum. För ett exempel se figur 6.1 och 6.2.

Frågan är hur grupperingen ska implementeras. Ett alternativ är att modifiera de metoder inuti Granska där parsning av matchningsregel nu sker. Att där lägga till möjligheten *group* och att implementera detta skulle bli en svår uppgift.

```
category kong {
info("kongruensfel")
link("" "")
}

kongruensfel@kong
{
    X(wordcl = dt),
    Y(wordcl = jj)*,
    Z(wordcl = nn & gender != X.gender)
-->
    group(X.gender = utr)
    group(X.gender = neu)
    corr(X.form(gender := Z.gender))
    action(scrutinizing)
}
```

Figur 6.1. *Matchningsregel innan gruppering.*

Ett annat alternativ, som inte påverkar Granskas inre strukturer, är att ta matchningsregeln och expandera den till flera regler enligt de grupperingsalternativ som angivits. Grupperingsvillkoren läggs sedan till som tilläggskrav på den sista ordvariabeln i varje expanderad regel. Rader med kommandot *group* tas sedan bort från reglerna och nu när reglerna är helt i överensstämmelse med ordinarie språksyntax körs de genom Granska (se figur 6.2).

För att kunna skilja rättelseförslagen åt innan utmatning till användaren utnyttjas att Granska för varje rättningsförslag har information om vilken regel som hittade felet. Vid expansionen av matchningsregeln ges de expanderade reglerna ett tillägg till sitt regelnamn. Vi får då ett sätt att skilja ut rättningsförslagen med hjälp av regelnamnen. Detta alternativ blev det som jag valde att implementera.

En ny funktion behöver nu implementeras som läser en matchningsregel från fil, parsar igenom den och om den innehåller kommandot *group* så expanderas den till flera regler. Resultatet blir en ny fil som sparas och som Granska sedan tar som inmatning.

Ett par saker måste utföras i funktionen: En regel har alltid en ”header” med information om *category* och liknande som Granska använder. Denna ”header” ska nu bara läggas in en gång, längst upp i filen, följd av de expanderade reglerna. Reglerna får också tillägg till regelnamnet. ”Originalregeln” `kongruensfel@kong` blir expanderad till reglerna `kongruensfel_group1@kong`, `kongruensfel_group2@kong` etc. Grupperingsinformationen för varje alternativ adderas sedan i motsvarande expanderad regel längst bak bland villkoren för den sista ordvariabeln. Här utnyttjas alltså också en egenskap som Granska redan har; att extra villkor för ordvariablerna kan läggas till den sista ordvariabeln. Regelns syntax och uppbyggnad måste självklart bevaras vid modifieringen.

```
category kong {
info("kongruensfel")
link("" "")
}

kongruensfel_group_0@kong
{
    X(wordcl = dt),
    Y(wordcl = jj)*,
    Z(wordcl = nn & gender != X.gender & (X.gender = utr))
-->
    corr(X.form(gender := Z.gender))
    action(scrutinizing)
}

kongruensfel_group_1@kong
{
    X(wordcl = dt),
    Y(wordcl = jj)*,
    Z(wordcl = nn & gender != X.gender & (X.gender = neu))
-->
    corr(X.form(gender := Z.gender))
    action(scrutinizing)
}

kongruensfel_group_2@kong
{
    X(wordcl = dt),
    Y(wordcl = jj)*,
    Z(wordcl = nn & gender != X.gender & (X.gender != utr &
    X.gender != neu))
-->
    corr(X.form(gender := Z.gender))
    action(scrutinizing)
}
```

Figur 6.2. *Matchningsregel efter gruppering. En regel har expanderats till tre. Regelnamnen har fått tillägg med grupperingsinformation. Grupperingsvillkoren har lagts till som extravillkor för ordvariabeln Z i respektive regel.*

Implementeringen av gruppering tillåter just nu bara uppdelning av *en* regel åt gången. I en utbyggd version är det bra om flera regler kan köras samtidigt från användaren, i så fall måste grupperingsfunktionen modifieras.

6.2 Slumpning och begränsning

I Granskas utmatningsfunktion (`PrintResult` i klassen `scrutinizer.cpp`) finns en for-loop där mening för mening tittas igenom. Endast de meningar som har fått ett rättelseförslag av regelmatcharen skickas till utmatning. Denna funktion använder jag i omdefinierad form `PrintResult_rl()` för att sortera grupperingsalternativen. Här finns också möjlighet att implementera både *slumpning* och *begränsning* på ett enkelt och effektivt sätt. Med hjälp av biblioteksrutinen `rand()` och en räknare bestäms nu i två extra kontroller vilka rättningsförslag som matas ut till användaren.

Slumpningsfaktorn får sitt värde av en variabel som skickas från användaren med sökfrågan. Nu gällande alternativ är 1/5, 1/10, 1/50, 1/100, 1/500, 1/1000. Största antalet utskickade rättningsförslag får också sin parameter från användaren. Alternativ: 10, 20, 50, 100, 500, 1000.

Båda dessa två faktorer innebär en begränsning av antalet rättningsförslag. Begränsningen skulle kanske kunna tillföras Granska i ett något tidigare skede, men det måste i alla fall vara efter regelmatchningen, när fel har upptäckts. Jag tycker att det här är det lämpligaste stället att lägga in ”extrakontrollerna”. Tidsförsämringen är minimal. Det blir två extra kontroller i konstant tid för varje rättelseförslag som Granska hittat.

Kapitel 7

Körning av prototypen

I detta kapitel redogör jag för testkörningar av prototypen med indexeringsförarbetet och med sök/ersättning mot ett par testkorpusar. Vidare går jag igenom de olika funktionsanropen till logiksidan och de svar till användargränssnittet som de ger upphov till. Jag redovisar också tidtagning av dessa funktioner.

7.1 Testkorpus

Jag har byggt testkorpusar i två omgångar. I den första omgången hämtade jag svenska exempeltexter från Internet. Jag försökte hitta texter med många grammatiska fel (vilket inte var så enkelt). För att överföra klartexten till korpusformat hade jag hjälp av ett par program som språkgruppen på Nada tillhandahöll. Först användes en tillämpning av Granska, *Tagger*, som taggar texten, sedan ett skript som fördelar den taggade texten till de olika filer som ingår i korpusformatet.

Vid överföringen omvandlas hela texten till gemener. Jag är osäker på om detta är korrekt. Ett resultat är att Granska inte kunde identifiera vissa förkortningar bestående av gemener, t.ex. taggades *ufo* som ett egennamn – en felaktig taggning. Däremot i ”original”-Granska taggas *UFO* (med versaler) korrekt.

Dessa texter samlade jag i tre testkorpusar. Jag lade också till egna planterade felaktigheter i texterna för att testa matchningsreglerna och funktionaliteten.

I den andra omgången har jag överfört 32 stycken texter från korpusen SUC till verktygets korpusformat. I korpusen SUC är texterna lagrade på SGML-format. På varje ny rad står ett ords *token*, *tagg* och *lemma* uppräknade.

Här hade jag också tillgång till hjälpprogram, ett Python-program och ett skript. I dessa bevaras versaler i överföringen, men jag fick andra problem med att analysera texterna i Granska. Granskas tagguppsättning grundar sig på SUCs tagguppsättning, men ett antal taggar är modifierade. Problemet var att Python-programmet inte ”översatte” dessa taggar till Granska-taggar. Efter att ha lagt till kod för konvertering av de vanligaste taggarna (som förekom i testtexterna) fungerar detta bra. En fullständig konvertering måste självklart göras, men tiden medgav inte detta.

För sökning i dessa testkorpusar se avsnitt 7.4.3.

7.2 Förarbete

Förarbetet med indexering av texterna fungerar bra och eftersom jag har kunnat testa den här programdelen separat så har programmeringsfel upptäckts tidigt och kunnat åtgärdas. Jag har försökt att få en bra felhantering med förklarande utskrifter för att underlätta arbetet för den systemansvarige.

Indexeringstiderna för några testkorpusar visas i tabell 7.1.

Tabell 7.1. *Indexeringstider för några olika korpusar. *Uppskattat tidsåtgång. Programmet indexerar ca 500 ord i sekunden.*

korpus	**antal texter**	**antal ord**	**indexeringstid**
test	5	7 467	16 s
test, *tåg*, *schack*	9	14 221	27 s
suc	80	191 000	5 min
stort*	500	1 000 000	30 min

Indexeringen tar relativt lång tid att utföra men det är acceptabelt med tanke på att det endast behöver göras då nya texter ska tillföras en korpus och bara över den då aktuella korpusen.

7.3 Parsning av inkommande XML-ström

Logiksidan har en serverfunktion som ligger och väntar på inkommande uppdrag. När ett uppdrag kommer ska XML-strömmen avkodas. Här fanns valet mellan en DOM-parser och en SAX-parser.

En SAX-parser, *Simple API for XML*, söker igenom data linjärt och anropar en eventhanterare varje gång den stöter på en tagg. En DOM-parser, *Document Model*, å andra sidan, läser data rad för rad och skapar ett träd som avspeglar den hierarkiska strukturen i XML-dokumentet. Dokumentet innehåller noder, där varje nod representerar ett element, ett attribut eller data.

En skillnad mellan dessa är att för SAX-parsern tas besluten om åtgärd direkt vid läsningen av XML-strömmen medan för DOM-parsern kan besluten tas när läsningen är färdig. En annan skillnad är att SAX-parsern tar konstant minne medan DOM-parsern tar linjärt minne i antalet trädnoder. Båda alternativen är tänkbara att använda. Båda alternativen har också nackdelen med en något invecklad implementation för att komma åt data. Jag valde DOM-parsern dels för att jag hade tillgång till exempelkod och dels för att implementationen blev något enklare.

7.4 Runtime

Vid testkörning av prototypen är det tre kommandon som är intressanta för logikdelen: *GetTexts*, *LoadCorpus* och *RunQuery*. Dessa har de förväntade svaren *Texts*,

Result respektive *Corrections*. När något exceptionellt inträffar används istället svaret *Error*.

För att få en uppfattning om exekveringstiden för olika kommandon har jag satt upp tidtagning på användarsidan från momentet precis innan ett kommando skickas iväg över nätverket tills momenten precis efter det att svaret tagits emot. Det allra första försöket i en serie tar alltid längre tid att utföra, sedan optimerar operativsystemet så att de följande går betydligt snabbare. Jag har valt att utelämna det första försöket och beräknat ett genomsnitt på de följande tio försöken för varje kommando.

Vid provkörning där både användardel och logikdel ligger på samma dator (där IP-adressen till logikdelen är `localhost`) tar t.ex. kommandot *GetTexts* 0,325 s att utföra. Antalet texter i katalogen var vid mättillfället ca 90. Som jämförelse provkörde jag med programdelarna på två olika datorer. Det gick något snabbare, exekveringstiden blev 0,318 s. För fortsättningen gäller tider för användardelen och logikdelen på två olika datorer.

Programdelarna använder mycket internminne. C++-programmet på logiksidan använder 27 Mbyte, korpus- och indexfiler (för ca 90 texter) 39 Mbyte, Java-Clienten 72 Mbyte och Java-servern 43 Mbyte. Java-sidan använder förvånansvärt mycket minne. En förklaring är att Javas virtuella maskin förbrukar mycket minne. Vid testkörning av ett litet Java-program användes hela 40 Mbyte. En annan förklaring kan vara att GUI:t som innehåller mycket grafik använder mycket internminne.

För att förtydliga har jag i detta avsnitt valt att inkludera bilder från GUI:t, dvs. från Helena Ihrfors del av examensarbetet.

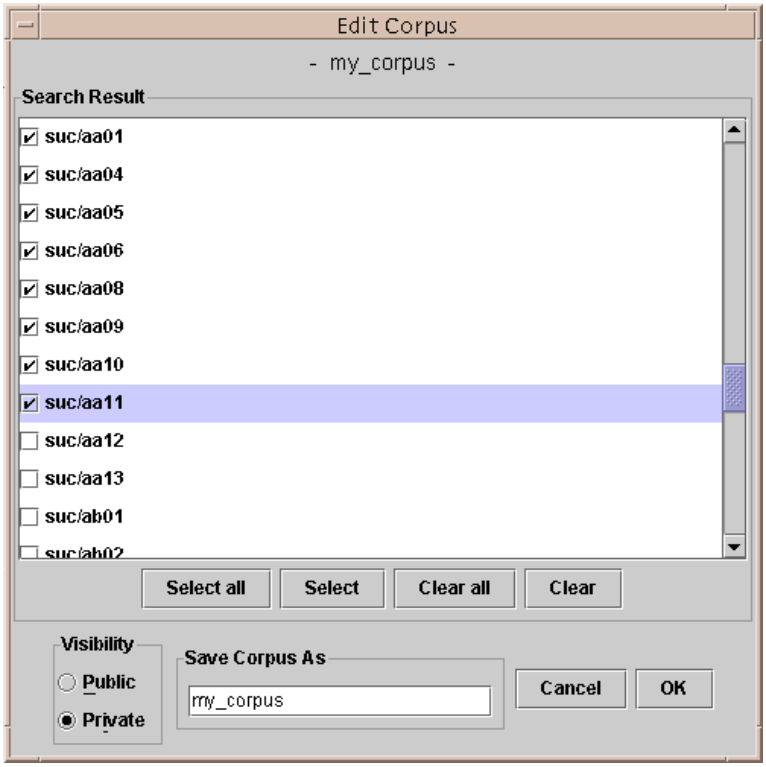
7.4.1 GetTexts

Användarsidan anropar logiksidan med *GetTexts* när en aktuell lista på samtliga korpusar med dess ingående texter behövs. Det sker vid menyvalen `Open Corpus` och `Edit Corpus`. Logiksidan traverserar då igenom katalogen *corpus* och listar där förekommande texter samt returnerar textinnehållet i en lista på XML-format. Texterna listas sedan i det öppnade dialogfönstret på användarsidan. Se figur 7.1 för ett exempel vid menyvalet `Edit Corpus`.

Som nämnts ovan tar kommandot *GetTexts* 0,325 s att utföra. Det är tillräckligt snabbt för att en användare inte ska uppleva någon fördröjning när dialogfönstret öppnas.

För att få ett mått på tidåtgången enbart för nätverksöverföringen ”hårdkodade” jag svaret *Texts* från logiksidan, som då direkt returneras. Svarsinnehållet var minimalt med endast ett textnamn. Körtiden blev nu 0,244 s. Det betyder att extratiden för att läsa de 90 textnamnen från filer och att skicka den större datamängden över nätverket är 0,081 s. För 500 texter borde tidsökningen bli ca 0,4 s. Det blir totalt en körtid på ca $0{,}6 - 0{,}7$ s, som kan upplevas som en liten fördröjning för användaren.

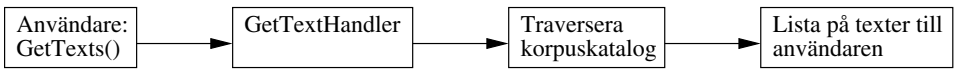
GetTexts är implementerad som en rekursiv traversering genom korpuskatalogerna vilket kan vara något tidskrävande. I en utbyggd prototyp kan man välja en annan implementering. Vid uppstart av logiksidan (efter t.ex. omindexering) låter



Figur 7.1. *Dialogfönster som öppnas i GUI:t vid menyvalet* `Edit Corpus`*. En lista över samtliga texter i alla korpusar presenteras. Användaren har valt att inkludera texterna "suc/aa01", "suc/aa04" etc. i den virtuella korpusen "my_ corpus".*

man en funktion göra denna katalogtraversering *en* gång och sedan spara resultatet på fil. Vid anrop från användarsidan returneras då bara innehållet i filen, vilket går mycket snabbt.

Händelseförloppet i *GetTexts* visas i figur 7.2.

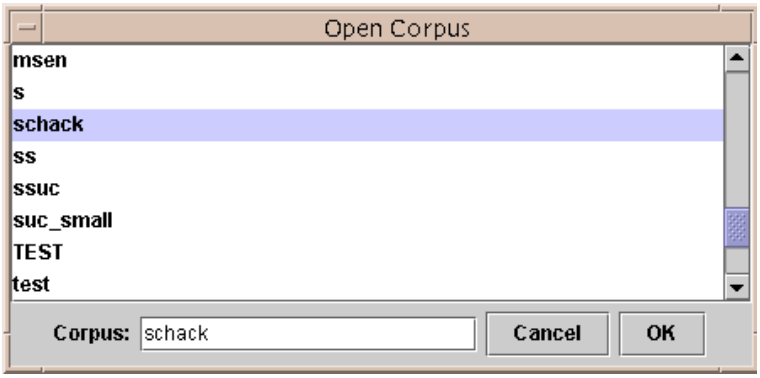


Figur 7.2. *GetTexts(), flödesschema.*

7.4.2 LoadCorpus

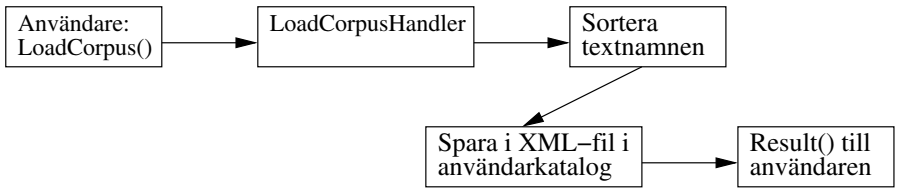
När en användare gör menyvalet `Load Corpus` anropas logiksidan med *LoadCorpus*. Det innebär att användaren definierar en egen virtuell korpus genom att en lista över valda texter skickas över till logiksidan, där den först sorteras för att sedan sparas på fil. Sortering görs för att underlätta senare hantering i andra programdelar. Se figur 7.3 för ett exempel på hur dialogfönstret i användargränssnittet kan se ut vid menyvalet `Load Corpus`.

Det här är ett enkelt och snabbt uppdrag och en användare märker knappast någon tidsfördröjning vid utförandet. Överföringstiden över nätverket samt utförandet på logiksidan ökar något för fler textnamn. Vid testning ”laddade” jag först en virtuell korpus bestående av 8 texter, detta tog 0,25 s att utföra. Då korpusen bestod av 90 texter hade tiden ökat till 0,27 s. En uppskattning ger då att 500 texter skulle kunna laddas på 0,35 s.



Figur 7.3. *Dialogfönster som öppnas i GUI:t vid menyvalet* `Load Corpus`*. Användaren har valt den virtuella korpusen ”schack”.*

Ett flödesschema över LoadCorpus visas i figur 7.4.



Figur 7.4. *LoadCorpus(), flödesschema.*

7.4.3 RunQuery

RunQuery anropas vid körning av en sökfråga. Användaren har angett en matchningsregel i regelfönstret i GUI:t; regeln kan innehålla kommandon med *group*, parametrarna *limit* och *random* skickas med.

På logiksidan skapas först ett *RunQueryHandler*-objekt som administrerar proceduren. XML-strömmen avkodas av en DOM-parser, matchningsregeln sparas på fil och övriga parametrar lagras. Matchningsregeln går sedan till en grupperingsfunktion, som tittar efter *group*-kommandon och om det finns något så expanderas regeln till flera regler. Regeln/reglerna laddas in i Granska, som parsar dem och utvinner optimeringsinformation. Optimeraren hämtar ut denna optimeringsinformation som är antingen en lista på taggbigram eller på sökord. Någon av dessa listor blir indata till motsvarande sökstruktur, som returnerar en lista av tupler med (textnamn/meningsspann). Dessa meningar läses från korpusfiler och läggs i en struktur som skickas till Granskas regelmatchare. Regelmatcharen genererar en lista med rättningsförslag. Till sist grupperas dessa och baserat på vad *random* och *limit* har för parametrar skickas rättningsförslagen till användaren. För ett exempel på hur körning av en sökfråga kan se ut i användargränssnittet, se figur 7.5.

Avvikande resultat är dels att inga meningsspann matchade optimeringsinformationen, dels att inga rättningsförslag genererades. Om något av dessa två fall inträffar används *Result* för att skicka över meddelandet `No matchings found` respektive `No corrections found` till användaren. Informationen visas på en statusrad i GUI:t.

Ett flödesschema över RunQuery visas i figur 7.6.

Några testkörningar för varierande antal texter, antal matchade meningar etc. visas i tabell 7.2.

Det som påverkar exekveringstiden är framför allt två saker. Den första är att det tar längre tid att söka i flera korpusar, eftersom programmet måste hämta data ur flera sökstrukturer. I fallet med taggbigram är detta märkbart eftersom man får så många träffar och eftersom dubbletterna måste gallras ut för varje sökstruktur. I fallet med ord får man inte så många träffar, men söktiden ökar med ca 0,15 s för varje extra sökstruktur.

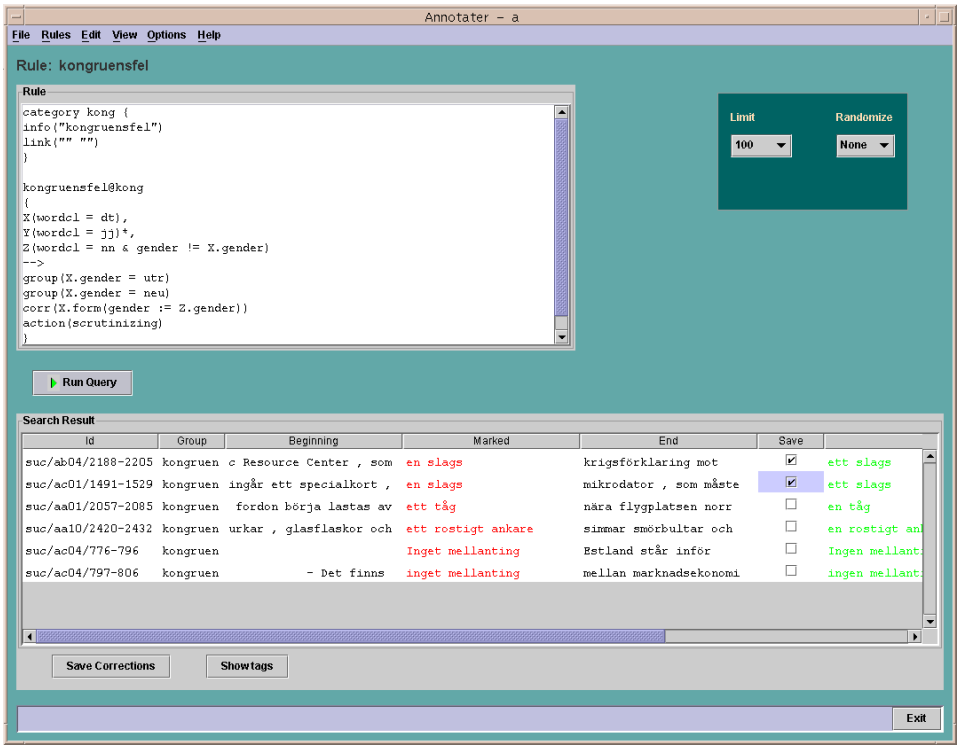
Den andra och största tidsfördröjningen sker när de aktuella meningsspannen hämtas in. Här ska programmet läsa från ett stort antal filer och samla ihop data i strukturer förberedda för granskning i regelmatcharen. Här märker man en klar skillnad i exekveringstid beroende på antalet träffar.

Ju mer preciserad en matchningsregel är desto färre träffar får man och desto snabbare går den att utföra. En virtuell korpus med texter från enbart en och samma korpus innebär också kortare exekveringstid.

Det skulle vara värdefullt att ha optimering på *taggtrigram* i en framtida version av verktyget, eftersom det skulle precisera sökandet och minska antalet träffar.

7.4.4 Felmeddelande

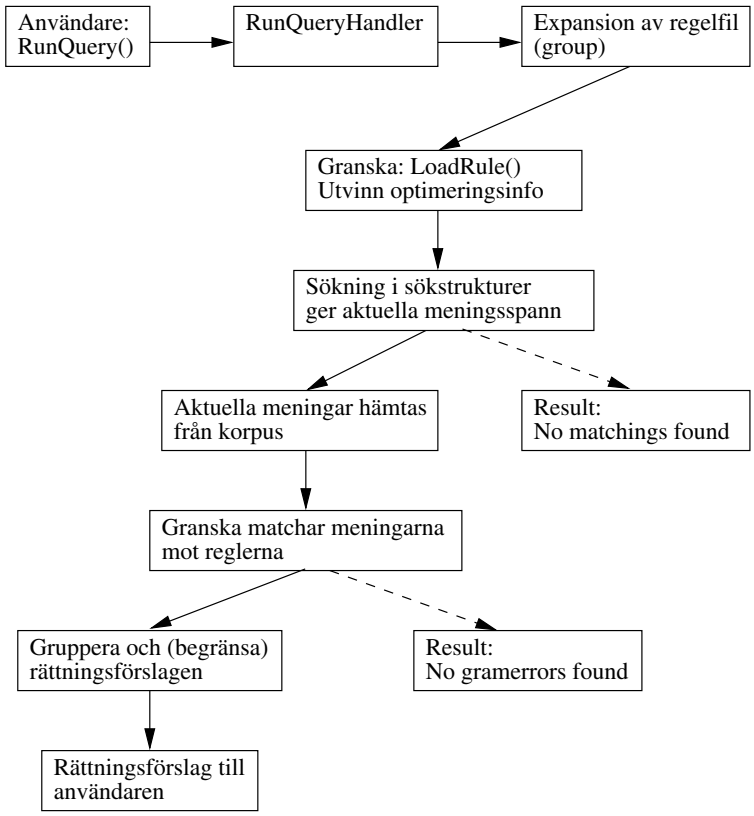
Då exceptionella händelser i programkörningen inträffar för något av de ovan uppräknade kommandona, används *Error* som returfunktion. Ett meddelande om felets



Figur 7.5. *Huvudfönstret i GUI:t (från gränssnittsdelen, Helena Ihrfors del). Uppe till vänster är regelfönstret, där matchningsregeln anges. Uppe till höger är två dropdown-menyer: "limit" och "randomize". Nertill är svarsfönstret som visar resultatet av en körning, där sex rättningsförslag genererades. I kolumnerna visas Id, grupperingsalternativ (delvis skymt), början på meningen, den felaktiga delen av meningen, slutet på meningen, en checkbox för att välja om rättelsen ska tillföras korpus eller inte och slutligen rättningsförslaget (delvis skymt). Användaren kan bestämma kolumnernas bredd individuellt genom att dra med muspekaren, det är också möjligt att rulla fönstret i sidled. Längst ner är statusraden (ej färdigimplementerad).*

art kommer då upp i ett modalt dialogfönster i GUI:t (användaren måste sedan trycka *OK* för att stänga fönstret).

För närvarande handhas bara vissa exceptionella händelser, bl.a. syntaxfel i matchningsregeln. I en färdigutvecklad version av verktyget måste självklart alla exceptionella händelser tas om hand.



Figur 7.6. *RunQuery(), flödesschema.*

Tabell 7.2. *Prestandatest för några körningar av sökfråga med olika angiven matchningsregel. Tiden (i sekunder) har mätts från det att Java-servern skickar kommandot över nätverket tills det att ett svar mottagits från logiksidan. Tiderna är genomsnittet av tio försök.*

texter	korpusar	opt.typ	matchade meningar	rättn.förslag	tid
1	1	ord	5	1	0,56
1	1	taggb.	4	2	0,55
5	1	taggb.	73	3	0,78
9	3	ord	3	2	0,59
9	3	taggb.	141	2	1,08
9	3	taggb.	190	8	1,31
10	1	taggb.	360	2	2,02
17	1	taggb.	567	3	3,10
32	1	taggb.	1093	11	7,56

Kapitel 8

Slutsatser och diskussion

I detta kapitel kommer resultatet av examensarbetet och implementationen av prototypen att diskuteras. Vidare presenteras slutsatser och rekommendationer för fortsatt arbete med verktyget.

8.1 Resultat

Arbetet har visat att utsikterna för att konstruera ett kraftfullt verktyg för sök- och ersättning i korpus är goda. Med hjälp av förindexering och optimering blir söktiderna rimliga. Med grupperingsfunktionen ges möjlighet för en användare (språkforskare) att på ett effektivt sätt godkänna/förkasta rättelseförslag som sedan kan tillföras korpus. Att på detta sätt kunna minska felprocenten i en träningskorpus som ett granskningsprogram som t.ex. Granska använder sig av, innebär att det kan få högre träffsäkerhet vid taggningen.

Verktygets uppdelning i två väl separerade delar där kommunikationen dem emellan går via socket/XML innebär en bra portabilitet. Korpustexter kan läggas upp centralt på en dator (server). Olika användare kan sedan ha sina applikationer med personliga inställningar på egen dator och kommunicera med servern via nätverket.

Det är förhållandevis enkelt att lägga till nya texter till korpuskatalogen och att indexera dem. För att överföra texter till korpusformat finns hjälpprogram att tillgå.

I nuläget finns optimering på ord eller taggbigram, vilket fungerar bra. Ett problem med taggbigramoptimering är att det är för allmänt eftersom det genererar ett stort antal träffar, som tar tid att behandla. Det vore önskvärt att också ha optimering på taggtrigram, som ger ett mindre och mer adekvat antal träffar.

Integreringen med Granska var en svår uppgift – här finns fortfarande problem att lösa. Denna del av implementationen tog lång tid att utföra och bidrog till att arbetet blev försenat.

Uppgiften var att konstruera en prototyp till ett verktyg. Att bara implementera grundläggande funktionalitet är en stor uppgift, många delproblem måste lösas under arbetets gång. Att till sist få allting att verkligen fungera rent praktiskt är en

krävande uppgift. Felhantering har implementerats till viss del men tiden har inte medgett att göra detta fullt ut.

8.2 Rekommendationer för fortsatt arbete

Det behövs mer arbete med kopplingen mot Granska för att få ett robustare system. Funktionerna i verktyget kan också optimeras ytterligare för högre prestanda. I rapporten anger jag ett flertal olika idéer för hur detta kan göras. Det som är mest tidskrävande är momentet med att hämta in de meningar från korpus som optimeringen vaskat fram för analys i Granska.

För att garantera att programmet är korrekt bör ett större test också genomföras. Valda texter kan testköras dels genom ”original”-Granska och dels överförda till korpusformat genom verktyget. Genom att testa Granska och verktyget på ett utvalt antal matchningsregler, en i taget, kan man då kontrollera att verktyget hittar och markerar rättningsförslag exakt likadant som Granska. Jag gjorde sådana tester på ett tidigt stadium av arbetet och då uppträdde allting normalt, men det bör självklart göras i större skala. Tester bör också göras med flera samtidiga användare.

Resultatet som helhet får betraktas som lovande men mycket arbete återstår givetvis till ett färdigt verktyg.

Referenser

Böcker

[1] Baase, S. & Van Gelder, A. *Computer Algorithms.* Third edition, Addison-Wesley, 2000.

[2] Ince, D. *Developing Distributed and E-commerce Applications.* Addison-Wesley, 2002.

[3] Knutsson, O. *Automatisk språkgranskning av text.* TRITA-NA-0105. KTH, Stockholm.

[4] Prata, S. *C++-programmering.* 3:e upplagan, Pagina.

[5] SAOL, *Svenska Akademiens ordlista över svenska språket.* Elfte upplagan. Norstedts förlag, 1986.

[6] *Svenska skrivregler.* Svenska språknämnden, Almquist & Wiksell, 1992.

Artiklar

[7] Bigert, J. *POS Tag Distance Metrics and Unsupervised Error Detection*, Nada, PTD, 2002.

[8] Bigert, J., Knutsson, O., Kann, V. och Sjöbergh, J. *Annotated Clauses and Flat Phrase Structures for Swedish*, Swedish Treebank Symposium, Växjö, november 2002.

[9] Carlberger, J., Domeij, R., Kann, V., Knutsson, O., 2000. *A Swedish Grammar Checker.*

[10] Carlberger, J. & Kann, V., 1999. *Implementing an efficient part-of-speech tagger.* Software – Practice and Experience, 29 (9), pp 815–832.

[11] Domeij, R., Knutsson, O. *Granskaprojektet: Rapport från arbetet med granskningsregler och kommentarer.* IPLab, Nada, KTH.

[12] Domeij, R., Knutsson, O., Öhrman, L., 1998. *Inkongruens och särskrivna sammansättningar.* Linköping Electronic Conference Proceedings. ISSN 1650–3686 (print), 1650–3740 (www).

[13] Kann, V. et al., 1999. *Stava.*

Webbsidor[1]

[14] BNC, *The British National Corpus.* http://www.hcu.ox.ac.uk/BNC/

[15] *CORBA website (success stories),* http://www.corba.org/

[16] *JNI-tips,* http://www.javaworld.com/javaworld/javatips/jw-javatip17.html

[17] *Jni-tutorial,*
http://java.sun.com/docs/books/tutorial/native1.1/stepbystep/index.html

[18] *Johnny/docs,*
http://www.nada.kth.se/∼johnny/docs/seminarie.020422.pdf

[19] *Myndigheternas skrivregler*
http://justitie.regeringen.se/klarsprak/sprakexperterna/handbocker/skrivregler/innehall.htm

[20] *Nada, theory*, http://www.nada.kth.se/theory/projects/granska/

[21] OMG: *OMG's CORBA FAQ.*
http://www.omg.org/gettingstarted/corbafaq.htm
http://www.omg.org/news/about/index.htm

[22] *Parole,* Svenska Parole. http://spraakbanken.gu.se/lb/parole/

[23] *Parole,* Tyska Parole. http://solaris3.ids-mannheim.de/parole.html

[24] *Stava,* http://www.nada.kth.se/∼viggo/stava/

[25] SUC *Stockholm-Umeå Corpus*, http://www.ling.su.se/DaLi/

[26] *Susning*, http://susning.nu/Korpus

[27] *Svarta listan*
http://justitie.regeringen.se/klarsprak/sprakexperterna/handbocker/svarta_listan.htm

Svarta listan innehåller en uppräkning av stela och formella ord och uttryck som bör undvikas.

[28] *Svenska datatermgruppen,* http://www.nada.kth.se/dataterm/

[1] Samtliga webbsidor besökta senast april 2003.

Kursdokumentation

[29] ADK och SUALKO *Föreläsningar och övningar 6*. Viggo Kann, 2001.

[30] *Internetprogrammering*, Internetprogrammering, 2001.
http://www.nada.kth.se/kurser/kth/2D1390/index.html