Datalogi C 2004, Roskilde Universitetscenter

Notat om syntaks og syntaksgenkendelse, særligt regulære udtryk og tilstandsmaskiner og lidt om anvendelser i bioinformatik

Henning Christiansen

November 2004

1 Indledning

Dette er en kort og hurtigt redigeret indføring til datalogiske metoder til leksikalsk analyse og syntaksgenkendelse og relaterede problemstillinger indenfor bioinformatik.

Det skal dog understreges at forfatteren er forsker og underviser i datalogi og først fornylig er begyndt at interessere sig for bioinformatik.

Der er ikke meget brødtekst, da der for det sprogliges vedkommende er vedhæftet uddrag af bogen

> H. Christiansen, Sprog og abstrakte maskiner, 278 pp., 3. reviderede udgave *Datalogiske noter*, 18 (3. udgv.), Roskilde Universitetscenter, 2000.

som iøvrigt kan anbefales til dem, som vil vide mere om emnet. Om regulære udtryk i Java henvises til tilgængelig on-line information og der gives blot et lille eksempel på anvendelsen.

Hvad angår det bioinformatiske, nøjes vi med nogle forenklede og hjemmegjorte eksempler, og der henvises til anden literatur og ressourcer tilgængelige på nettet.

2 Kort om syntaks og syntaksanalyse; læsevejledning til uddrag af *Sprog og abstrakte maskiner*

Syntaksanalyse ud fra formaliserede grammatikker er en gammel datalogisk disciplin, der er lige så gammel som idéen om programmeringssprog og compilere. I stedet for at programmere en computer vha. 0-er og 1-er, skriver man sine programmer i noget, som er lidt nemmere at forstå for mennesker. I tidernes morgen i form af maskinsprog i stil med

```
MOVE 123 3564
JMP 111
ADD 444 @2
```

eller strukturerede programmeringssprog som Cobol, Algol60, Pascal eller Java. F.eks.:

```
for(int j=0; j<= s2.length()-windowSize; j++)
  {for(int i=0; i<= s1.length()-windowSize; i++)
     {int commonChars = 0;
      for(int k=0; k<windowSize; k++)
        {if(s1.charAt(i+k)==s2.charAt(j+k)) commonChars++;}
      if(commonChars>=cuttOff) mark(i,j);}}
```

Man skelner mellem *leksikalsk syntaks* og *strukturel syntaks* for et programmeringssprog (betegnelserne kan variere).

Det leksikalske niveau handler om udseendet af de simpleste elementer så som tal og identifiers. F.eks. ved vi i Java, at `1234` kan gå som et heltal og `sodaVand` som en identifier (f.eks. anvendt som navn på en variabel).

Leksikalsk syntaks beskrives som regel ved *regulære udtryk*, som er beskrevet i *Sprog og abstrakte maskiner* (fremover refereret til som SAM) afsnit 4.2.2. For eksempel kan ordinære heltal beskrives ved følgende udtryk,

```
(1+2+3+4+5+6+7+8+9+0)+
```

Her er plusser og paranteser metasymboler, og øvrige tegn repræsenterer sig selv. Infix plus svarer til foreningsmængde, paranteser til gruppering, og det efterhængte (postfix) plus svarer til repetion én eller flere gange. Definition og flere eksempler i SAM.

Vi taler om "sprog" i en søgt og formaliseret betydning, og et sæt regulære udtryk definerer et bestemt sprog, som er en mængde af strenge (= sekvenser af tegn). Udtrykket ovenfor bestemmer sproget svarende til det, vi tænker

på som mængden alle mulige ikke-negative heltalskonstanter. Sådanne sprog kaldes *regulære sprog*.

Man kan gøre rede for, at et sprog som Java ikke er regulært, dvs. ikke kan beskrives ved regulære udtryk alene. Det grundlæggende problem er de regulære udtryks mangel på rekursion, og et minimalistisk eksempel på et sprog, som ikke er regulært er mængden af strenge, som består af først et antal `a`'er og så lige så mange `b`'er. For eksempel er `aaabbb` med, men ikke `aaaaab`; det regulære udtryk `(a+)(b+)` beskriver godt nok sekvenser af `a`'er efterfulgt af `b`'er, men det kan, om man så må sige, ikke "huske" hvor mange `a`'er der blev set, før vi kommer til `b`'erne.

Normalt beskriver man den strukturelle syntaks af et sprog som Java ved en *kontekst-fri grammatik*. En sådan beskrivelse er dog ikke komplet, da man ikke med en kontekst-fri grammatik kan indfange det aspekt, at metoder, variable og klasser m.v. skal benyttes i overensstemmelse med deres erklæringer og typer.[1]

Kontekst-fri sprog kan beskrives vha. *BFN-notation* eller *syntaks-diagrammer*, som også er beskrevet i SAM, afsnit 4.2.2. Følgende regel beskriver strenge af `a`'er efterfulgt af så lige så mange `b`'er.

⟨ok-streng⟩ ::= `ab` | `a` ⟨ok-streng⟩ `b`

Tag igen et kig på fragmentet af et Java-program som blev vist ovenfor. Et væsentligt træk er, at de krøllede paranteser skal matche, og her kan vi se analogien til `ab`-eksemplet, hvor hvert `a` skal matches med et `b` længere henne i strengen, og det som står imellem skal overholde tilsvarende krav (rekursivt indad) for sine `a`'er og `b`'er.

Regulære udtryk og kontekst-fri grammatikker er beskrivelser af sprog, men de siger i princippet intet om, hvordan man analyserer den en eller anden tekststreng udfra en grammatisk beskrivelse af den ene eller anden art. Denne form for analyse kan defineres i den mest enkel form som det at sige ja eller nej til om en streng er med i sproget, men det er mere interessant, hvis analysen resulterer i en repræsentation af det som er genkendt, f.eks. i det kontekst-fri tilfælde i form af et syntakstræ.

Regulære udtryk kan skrives om til såkaldte endelige tilstandsmaskiner (ofte blot kaldet tilstandsmaskiner), der kan forstås som abstrakte algoritmer til analyse; vedlagte kapitel 12 af SAM beskriver dette i detaljer og hvordan man kan implementere tilstandsmaskiner som et program.

Analyse af kontekst-fri sprog kaldes i fagjargon *parsing*, og der findes et større udvalg af metoder til formålet; vedlagte kapitel 13 fra SAM beskriver et udvalg.

[1] Det er ikke helt forket at sige at det kontekst-*fri* netop ligger i det at man ignorerer konteksten givet ved erklæringerne.

I praksis benytter man ofte programgeneratorer til at konstruere de programmer, som foretager analysen. Det kan f.eks. fungere ved, at man skriver en input-fil som indeholder regulære udtryk (eller en kontekst-fri grammatik), og vips, så produceres en tekstfil med et program, som kan analysere tekster. Vil du vide mere, er denne bog uundgåelig:

> A.V. Aho, R. Sethi og J.D. Ullman, *Compilers, Principles, Techniques and Tools.* Prentice-Hall, 1986.

Til sidst skal det nævnes at tilstandsmaskiner faktisk kan benyttes, og benyttes i stor stil indenfor datalingvistikken, til sprog, som overstiger niveauet for regulære sprog.

Meget kort forklaret, så starter man med at udstyre tilstandsmaskiner med et output, som produceres mens de gnasker sig igennem en inputstreng, og på den måde opnås der et oversættelse af inputstrengen; en sådan udvidet tilstandsmaskine kaldes ofte en *transducer*. Man kan nu f.eks. lade en første transducer æde sig igennem en inputstreng, og skrive den ud igen dekoreret med ekstra information, og tricket er nu at at lade en anden transducer æde sig igennem den streng som blev produceret af den forrige transducer. Generelt taler man om *cascading*, når man sætter et antal transducere efter hinanden. Søg selv mere litteratur, hvis det lyder spændende.

3 Om regulære udtryk i Java

Programmeringssproget Perl er konstrueret med henblik på strengmanipulation og indeholder regulære udtryk og strengmanipuleringsfaciliteter som grundlæggende elementer i sproget. Perl benyttes bl.a. til boinformatik og datalingvistik og også til netværksprogrammering (hvor mans skal finde hovede og hale i datapakker modtaget over en transmissionslinje). Iøvrigt bygger Perl videre på sproget SNOBOL, som kan spores tilbage til 1960'erne.

Java API indeholder faciliteter inspireret af Perl i pakken `java.util.regex`. Se et af følgende steder afhængigt af, hvilken version af Java du bruger:

- http://java.sun.com/j2se/1.4.2/docs/api/index.html
- http://java.sun.com/j2se/1.5.0/docs/api/index.html

Pakken er delt i to klasser, `Pattern`, som definerer en notation for regulære udtryk, som er væsentlig udvidet i forhold til den beskrevet i uddraget af SAM, og `Matcher`, som indeholder faciliteter til strengmanipulation, herunder naturligvis til matching af strenge ud fra givet `Pattern`.

Desuden har version 1.5.0 en klasse `Scanner`, som også benytter notationen for regulære udtryk, men det er ikke klart hvor den spiller sammen med eller overlapper `regexp`.

Der har ikke været tid til at udvikle pædagogiske eksempler på dette sted, men der vil blive stillet opgaver i at bruge `Pattern` og `Matcher`.

4 En sproglig indfaldsvinkel på Bioinformatik

Det skal understreges at forfatteren til dette notat er en *absolut nybegynder* indenfor alt hvad der hedder bioinformatik, og det som præsenteres nedenfor må på ingen måde tages som andet end amatøragtig kradsen i overfladen.

Som introduktion til emnet henvises til en bog, som også har været inspirerende for det som følger.

> D.E. Krane, M.L. Raymer: *Fundamental concepts of Bioinformatics* Benjamin-Cummings, 2003.

Følgende website tilbyder et udvalg af Java-programmer til bioinformatik:

> http://www.biojava.org/

En bog som også kan anbefales, er

> *Artificial Intelligence and Molecular Biology*, Lawrence Hunter (red.). AAAI (årstal ikke kendt)

Den er udsolgt fra forlaget, men er frit tilgængelig på denne adresse:

> http://www.aaai.org//Library/Books/Hunter/hunter.html

Specielt disse kapitler kan anbefales som en indgang til emnet:

1. Molecular Biology for Computer Scientists, Lawrence Hunter.
2. The Computational Linguistics of Biological Sequences, David B. Searls.

4.1 Biokemi som sprog

Gener og proteiner og den slags er opbygget udfra aminosyrer, og der er fire som er væsentlige. Disse forkortes C, G, A og T (og hvad det står for er os inderligt ligegyldigt for nærværende). En gensekvens eller et protein kan betragtes som en lang streng af disse aminosyrer, hvor det siger sig selv, at man har abstraheret en masse yderligere information væk. Men med tilpas

snæversyn kan man påstå at genetik handler om strenge over et alfabet med fire bogstaver.

Det er således oplagt, at de tekstsøgningsalgoritmer, som vi kender fra datalogien baseret på regulære udtryk/tilstandsmaskiner, må kunne anvendes, f.eks. Knuth-Morris-Pratt, jvf. SAM kap. 14. Tilsvarende med parsealgoritmer til indetifikation af mere komplicerede strukturer. Dog er problemstillingerne ofte en smule anderledes, og der udviklet et hav af specialiserede algoritmer.

Når man studerer, hvordan forskellige gener har udviklet sig over tid, kan man f.eks. sammenlige en "oprindelig" sekvens med én som man tror er en efterkommer. Man kan da sammenligne bogstaverne, se om der er kommet nye gener ind i sekvensen eller nogle har ændret sig. Betragt f.eks.

1. AGATCATCAT
2. CGATCATAGCAT

Her kunne man få det indtryk at streng 2 er fremkommet ved ændring af 1 ved, at det første A er muteret til et C og der er smuttet AG ind før sidste CAT.[2] Beregningsmæssigt kan dette være meget kompleks, og strenge kan være hundrede eller tusinder af bogstaver lange.

En ofte forekommende problemstilling er, at man har en stor samling strenge svarende til forskellige mutationer af en bakterie eller en ondsindet virus, man gerne vil have bugt med. Her vil man gerne opstille de forskellige udgaver i et slags familietræ, så man kan se, hvem der nedstammer fra frem. Dvs., man har en stor mængde af strenge som man skal sammenligne parvis to ad gangen for at kontrollere, om den ene mon kunne nedstamme direkte fra den anden. — Her bemærker det erfarne udi algoritmekunsten, at dette giver en faktor n^2, hvor n er antallet af strenge, som bliver ganget på den allerede høje kompleksitet, vi observerede før.

På grund af den store samfundsmæssige interesse i sagen er der udviklet et stort udvalg af algoritmer til formålet, og der findes adskillige datamængder og analyseværkøjer frit tilgængeligt på internettet.

4.2 Dot-Plot-teknikken

En måde at komme igang med at sammenlige to gensekvenser kan være at undersøge om der forekommer stumper (gener), som gentages i de to strenge, men måske ikke i helt samme rækkefølge, og måske udsat for mindre mutationer (som en datalog kunne fristes til at kalde mindre syntaksfejl).

[2]Der er muligvis noget om at de aminosyrer ligger i grupper af et vist antal; det (og en hel masse andet) ser vi bort fra her!

Problemet lyder umiddelbart meget kompliceret (og komplekst i datalogisk forstand), men der findes en indlysende og genial måde at at gå løs på det, og den kaldes Dot-Plot (Den er indlysende, når man får den forklaret og ser et eksempel...). Vi har altså to strenge, og den ene (streng 1) skriver vi ad af en x-akse og den anden (streng 2) opad en y-akse. Det definerer et koordinatsystem, og i de koordinatpar (x, y) hvor streng 1 har samme bogstav på x'te plads, som streng 2 har på y'te plads, sætter man et prik.

Fælles delstrenge tegner sig som skrå streger i det færdige billede. Forvirret? Her følger et eksempel. Lad de to strenge være

1. `abe---abe---abekat---`
2. `kat...abekat...`

Dot-plotten for de to ser således ud (hvor vi bruger `x` i stedet for en prik).

```
.
.
t                 x
ax     x     x   x
k               x
e  x     x     x
b x     x     x
ax     x     x   x
.
.
.
t                 x
ax     x     x   x
k               x
 abe---abe---abekat---
```

Den længste skrålinie svarer til den dobbelte forekomst af `abekat`, og de mindre skrålinie svarer til kortere delstrenge, ned til enkelt-bogstaver som også går igen.

For biologen vil det være sekvenser af ACGT, og han eller hun vil ud fra sin faglige indsigt og den aktuelle problemstilling kunne sige noget om, hvor lange fælles delstrenge skal være for at de kan kaldes interessante, og også hvor mange afvigelser, der kan accepteres.

Man kan trimme algoritmen ved at indføre to parametre kaldet `windowSize` og `cutOff`. Vi sætter nu en prik i (x, y), såfremt der i de næste, efterfølgede `windowSize` tegn er mindst `cutOff` som stemmer overens i de to strenge. Betegnelsen `windowSize` kommer af, at man kan tænke på det som om man lader et vindue af den angivne længde glide henover de to strenge.

I eksemplet overfor kan det være at delstrenge skal være 5 tegn eller derover for at have betydning, men hvor vi tillader et enkelt tegn inden for de 5 at afvige; det giver `windowSize=5` og `cutOff=3`. Vi indfører en enkelt afvigelse i streng nr. 2 (skriver `abøkat` istedet for `abekat`), og kører algoritmen igen:

```
.
.
.
t
a
k
?
b              x
a             x
.
.
.
t
a
k
 abe---abe---abekat---
```

Skrålinjen bestående af to prikker indikerer en fælles delstreng af længde $2 + 5 - 1 = 6$ tegn, hvor der (som tilfældet er her) kan forekomme en enkelt afvigelse. Et program til Dot-Plot kan skrives ved nogle få linier Java-kode og gengives her i sin fulde længde.

```
class DotPlot{
 static String s1 =  "abe---abe---abekat---";
 static String s2 =  "kat...abøkat...";
 static char [][] plot = new char[s1.length()+1][s2.length()+1];
 static final int windowSize = 5;
 static final int cuttOff = 4;

 static void initPlot()
   { plot[0][0]=' ';
     for(int i=0; i< s1.length(); i++) plot[i+1][0]=s1.charAt(i);
     for(int j=0; j< s2.length(); j++) plot[0][j+1]=s2.charAt(j);
     for(int j=1; j< s2.length(); j++)
       {for(int i=1; i< s1.length(); i++) plot[i][j]=' ';}}

 static void mark(int i, int j) {plot[i+1][j+1]='x';}

 static void printPlot()
    { for(int j=s2.length(); j>= 0; j--)
        {for(int i=0; i<= s1.length(); i++)
           {System.out.print(plot[i][j]);}
         System.out.println();}}

 public static void main(String [] args){
      initPlot();
      for(int j=0; j<= s2.length()-windowSize; j++)
        {for(int i=0; i<= s1.length()-windowSize; i++)
           {int commonChars = 0;
            for(int k=0; k<windowSize; k++)
              {if(s1.charAt(i+k)==s2.charAt(j+k)) commonChars++;}
            if(commonChars>=cuttOff) mark(i,j);}}
      printPlot();}}
```

Læs mere om algoritmen i Krane & Raymers bog, eller en af de mange andre bøger om bioinformatik som vælter frem på markedet for tiden.

4.3 At regne sig frem til formen af et protein ud fra dets aminosekvens

Proteiner har det med at sno og dreje sige på den mest finurlige vis, og man kan til en vis grad regne sig frem til formen ved at undersøge sekvensen af aminosyrer. Vi vi se her på et enkelt fænomen, som er løkker. Et protein er beskrevet ved en lang gensekvens, og hvis der er to delsekvenser som særligt tiltrækkes af hinanden, så kan der opstå en løkke. Denne tiltrækning er

elektrisk m.v., modsatte ladninger tiltrækker hinanden, og ud fra yderligere viden om den kemiske struktur ved man at G og C særligt tiltrækker hinanden og tilsvarende for A og T.[3]

For en givet delstreng siger vi at dens *modsatte streng*, er den som kommer ved at

- erstatte alle G med C,
- alle C med G,
- alle A med T,
- alle T med A,
- og så skrive resultatet op baglæns.

Tag for eksempel strengen AGACAT. Først udskifter vi bogstaverne og får TCTGTA, og skrives den baglæns får vi altså den omvendte streng ATGTCT. Lad nu s og t stå for delstrenge og lad $\bar{s}$ stå for den omvendte streng for s. En løkke kan opstå hvis der i en streng forekommer følgende mønster $\cdots st\bar{s}\cdots$. Vi har ikke lige et godt tegneprogram ved hånden så læseren må forestille sig at at proteinet ligger og slanger sig, så s og $\bar{s}$ klæber op ad hinanden, og når de gør det, så vil t blive krummet sammen og danne en løkke.

Vi kan altså beskrive fænomenet løkke i form af en grammatik-regel

$$\langle \text{løkke} \rangle ::= \quad \langle \text{sekvens}_1 \rangle \ \langle \text{sekvens}_2 \rangle \ \langle \overline{\text{sekvens}_1} \rangle$$

Det går ganske vist en tand videre end det, vi tillader i en kontekst-fri grammatik, men definitionerne lader sig nemt generalisere til formålet.[4] Andre, mere komplicerede syntaksregler med kombinationer af flere omvendte strenge kan beskrive figurer som tre- og fir-kløvere o.m.a. Der er så yderligere nogle tommelfingerregler om længderne på de involverede delstrenge for at de forskellige figurer forventes at opstå, men det ser vi helt bort fra her.

Det at analysere en lang proteinstreng for at finde kombinationer af en streng og deres omvendte er ekstremt kompleks, også fordi der kan være løkker i løkker osv., og der kan også være flere forskellige måde at fortolke strengen på (dvs. flere måder at parre de forskellige match på). Algoritmer til realistiske anvendelser går langt udover det vi kan få plads til her, så vi nøjes med nogle små ekserimenter, som viser en lille smule om hvad problemet går ud på.

[3]Det anbefales at kan ikke skriver dette ukritisk af, men checker efter i en bog skrevet af fagfolk.

[4]Søg efter *"attribute grammars"*, D.E. Knuth.

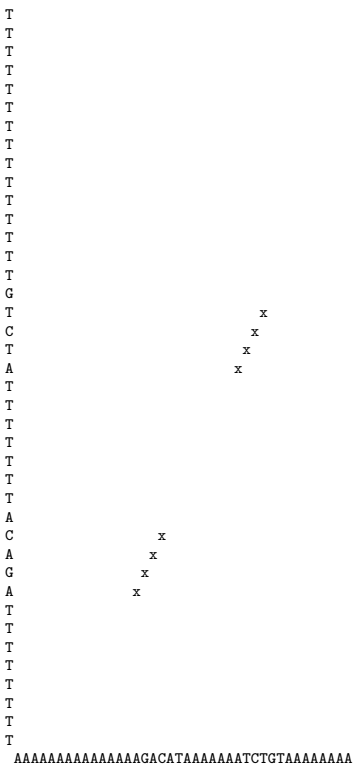
Et første bud på at identificere mulige kandidater til "klister-delstrenge" kan vi få ved at benytte Dot-Plot-metode. Lad s betegne denne streng, hvor den opmærksomme læser vil se at vi omhyggeligt har indkodet en enkelt løkke vha. AGACAT og dens omvendte ATGTCT.

AAAAAAAAAAAAAAAGACATAAAAAAATCTGTAAAAAAAA

Tag nu det omvendte af hele denne streng, $\bar{s}$, som ser således ud.

TTTTTTTTAGACATTTTTTTTTCTGTTTTTTTTTTTTTTT

Et Dot-Plot med `windowSize=cuttOff=3` ser således ud.



Dette afslører tydeligt hvor en løkke kan forekomme. Hvis der er mange match (hvilket der vil være for enhvert realistisk eksempel), man man så afmærke dem i strengen, så man får en mere abstrakt repræsentation, som kunne minde om f.eks. dette her:

$$\cdots s \cdots t \cdots \bar{s} \cdots \bar{t} \cdots$$

Dvs. man afmærker matchene og noterer resten som uinteressant i denne sammenhæng. Prøv som en øvelse at tegne hvilken figur denne konstellation mon kan give anledning til.

Når vi skal karakterisere et metasprog, kan vi gentage begreberne syntaks, semantik og pragmatik, og alt hvad vi har nævnt derunder. Når det drejer sig om metasprog, er det f.eks. nødvendigt at se på, hvilken måde abstrakt og/eller konkret syntaks for et objektsprog kan repræsenteres, og vi kan se på, hvor nemt (eller besværligt) det er at skrive fortolkere eller oversættere.

4.2 Metasprog

4.2.1 Pragmatisk diskussion omkring metasprog

Et metasprog er et sprog i hvilket man udtrykker egenskaber om det sprog, nu engang er det aktuelle objektsproget. Et metasprog må rumme en repræsentation af større eller mindre dele af objektsproget. Hvor nogle metasprog eksempelvis kun vedrører leksikalsk syntaks, er der andre metasprog, som retter sig mod objektsprogets samlede syntaks og semantik. I det sidste tilfælde må vi forvente, at der findes (eller kan udtrykkes) repræsentationer af leksikalske symboler, af abstrakte (og måske også konkrete) syntakstræer, samt måder at specificere semantikken, f.eks. ved en fortolker eller oversættelse — eller hvis metasproget er af en rent beskrivende (dvs. ikke processerbar på interessant måde af en maskine), måske ved passende matematiske objekter.

I det følgende ser vi først på beskrivelsessprog for syntaks, dernæst diskuterer vi kort beskrivelsessprog for semantik; i de efterfølgende afsnit vises, hvordan Prolog kan benyttes som metasprog for syntaks og semantik.

4.2.2 Klassiske metasprog for syntaks

Regulære udtryk

Regulære udtryk er et værktøj til beskrivelse af mængder af tekststrenge med en ikke for kompliceret struktur, og er således velegnede i forbindelse med leksikalsk syntaks. Eksempelvis kan man beskrive, hvordan reelle tal skrives i et programmeringssprog, men udtryk med indlejrede parenteser ligger uden for de regulære udtryks domæne.[6] Vi definerer regulære udtryk induktivt på følgende måde; vi går ud fra, at et *alfabet*, dvs. en mængde af tegn (f.eks. *a*, *b*, *c*, …) er givet på forhånd.

[6] Regulære udtryk beskriver sprog af type 3 i Chomsky-hierarkiet, hvor de indlejrede udtryk hører hjemme i type 2, de kontekstfri sprog (Chomsky, 1959). De sidstnævne kan beskrives ved syntaksdiagrammer, som svarer til tilstandsmaskiner (eller regulære udtryk) med den ekstra egenskab at kunne referere rekursivt til hinanden. Eller alternativt, en såkaldt »push-down automaton«, som er en tilstandsmaskine udvidet med en stak.

– ϵ er et regulært udtryk, som beskriver den tomme streng.

– for et givet tegn, *a*, så er *a* et regulært udtryk, som beskriver strengen bestående af *a*.

– hvis A og B er regulære udtryk, så er AB et regulært udtryk som beskriver mængden af strenge, som fremkommer ved at tage en streng beskrevet af A og sætte sammen med en streng beskrevet af B.

– hvis A og B er regulære udtryk, så er A+B et regulært udtryk som beskriver foreningen af strenge beskrevet af A og af B.

– hvis A er et regulært udtryk, så er A^* et regulært udtryk, som beskriver mængden af strenge, som forekommer ved at sammensætte 0, 1, 2, …strenge, hver især beskrevet af A,

– A^+ er defineret som A^*, men her skal være mindst 1 delstreng.

Parenteser kan bruges på sædvanlig måde til at gruppere, og der gælder sædvanlige konventioner for at undgå for mange parenteser, eksempelvis:

– AB^* læses som $A(B^*)$

– $AB+C^*D$ læses som $(AB)+(C^*D)$.

Som det fremgår, udgør regulære udtryk en nydelig algebra, hvor der kan opstilles forskellige regneregler, f.eks.

– $A(B+C) = AB + AC$

– $A\epsilon = \epsilon A = A$

– $A^+ = AA^*$

Som gennemgående eksempel benytter vi følgende, som er hentet fra (Sedgewicks, 1988). Udtrykket

$$(a^* + ac)d$$

beskriver en mængde af strenge, som består af:

bd, abd, aabd, aaabd, aaaabd, …, acd.

Ofte benytter man forskellige udvidelser til regulære udtryk, f.eks.

– skriver *ciffer* i stedet for 0+1+2+3+4+5+6+7+8+9,

– skriver *bogstav* for *a*+…+*å*+*A*+…+*Å*.

– skriver [A] for ϵ + A (læses: eventuelt A).

Dette skal bemærkes, at der blot er tale om en udvidelse af notationen og ikke af formalismens udtrykskraft.[7]

Det skal dog understreges, at en samling regulære udtryk ikke er nok til at beskrive den leksikalske syntaks af et sprog, da det også skal specificeres, hvorledes de leksikalske symboler skelnes fra hinanden. Betragt f.eks. tegnsekvensen `begin`. Hvis den optræder i et program skrevet f.eks. i Pascal og er omgivet af mellemrum eller linieskift, så er der tale om et reserveret kodeord og ikke om de to variable `be` og `gin` placeret tæt op af hinanden. Det vil ofte være tilstrækkeligt med følgende konventioner:

– Hvert leksikalsk symbol udspænder en så stor delstreng som muligt,

– blanktegn og linieskift er tilladt mellem leksikalske symboler og ignoreres iøvrigt.

Lad os som eksempel betragte aritmetiske udtryk, svarende til en forenklet udgave af dem, vi kender fra de sædvanlige programmeringssprog. Variable er sekvenser af bogstaver og cifre, startende med et bogstav:

bogstav(*bogstav*+*ciffer*)*.

Vi har her kun simple heltal uden fortegn: *ciffer*$^+$. Øvrige leksikalske symboler er beskrevet ved regulære udtryk, bestående af et enkelt tegn og svarer til regneoperatorer og paranteser: »+«, »-«, »*«, »(«, »)«. Bemærk, at vi her benyttede anførselstegn for at kunne skelne mellem hvad der er objektsprog, og hvad der er metasprog. Anførselstegnede fungerer her som en del af metasproget, og eksempelvis plusset mellem anførselstegnene er et symbol fra objektsproget, således at »+« bliver (konkret syntaks for) en frase i metasproget hørende til den syntaktiske kategori »et tegn«.

Et andet problem, som optræder i forbindelse med leksikalsk syntaks er, at visse tegnsekvenser har særlige roller som reserverede ord, f.eks. `begin`, `end`, `while` osv., hvor så alle tegnsekvenser bortset fra disse kan benyttes som identifiers. Det er teknisk muligt, men meget besværligt, at konstruere et regulært udtryk svarende alle tegnsekvenser pånær et givet udvalg, og derfor er det hensigtsmæssige at udvide metasproget med formuleringer, som involverer »bortset fra«.

[7] Altså helt analogt til begrebet syntaktisk sukker indenfor programmeringssprog.

Opgave 4.3 Hvor mange lag af sprog og metasprog er involveret i tekstafsnittet »Et andet … fra«.« ovenfor? Og hvad med denne opgavetekst?

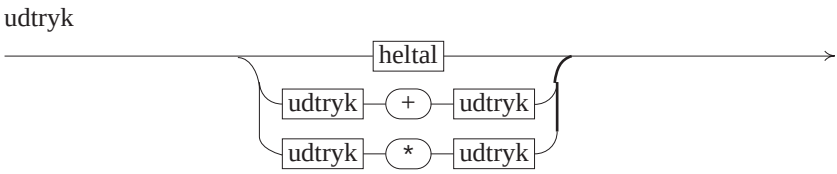
Opgave 4.4 Beskriv et regulært udtryk for tal i et programmeringssprog, hvor følgende er eksempler på tal: 1, 3.14, -1.217e35.

Opgave 4.5 Skriv et prædikat i Prolog, som givet en tekststreng og et regulært udtryk undersøger, om strengen er omfattet af udtrykket.

BNF og syntaksdiagrammer

Vi beskriver som regel strukturel syntaks ved en *kontekst-fri grammatik*, som består af en række *terminalsymboler* (eller leksikalske symboler) og *nonterminalsymboler* (kort nonterminaler) samt et antal *syntaksregler* eller *syntaksdiagrammer*, som forklares ved eksempler nedenfor.

Vi vil diskutere dette i forhold til sprog af aritmetiske udtryk, hvis abstrakte syntaks, vi beskrev tidligere. Lad os antage passende leksikalske symboler defineret for sproget, og vi vil i første omgang skrive den abstrakte syntaks naivt om til et syntaksdiagram, som angiver en konkret, strukturel syntaks. Her har vi kun en nonterminal, nemlig »udtryk« svarende til kategorien fra den abstrakte grammatik. Til hver nonterminal svarer ét diagram, og nonterminaler kan refereres rekursivt inde fra et diagram, hvor nonterminalen angives ved en firkantet kasse. Terminalsymboler optræder i afrundede kasser, dog med den undtagelse at hvis der er tale symboler defineret ved et ikke-trivielt regulært udtryk, optræder de, som var de nonterminaler (med det rationale, at det er defineret andetsteds, f.eks. ved et regulært udtryk).

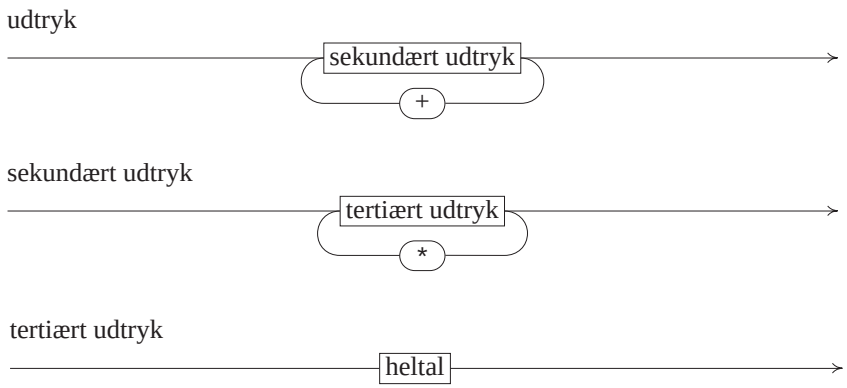


En kontekstfri grammatik beskrevet ved et eller flere syntaksdiagrammer karakteriserer en mængde af sekvenser af terminalsymboler, defineret som følger. For givet nonterminal lokaliserer vi det tilhørende diagram og følger en vej gennem dette startende fra øverste venstre indgang. Møder vi et navn i en firkantet kasse, refererer det til et andet diagram (eller regulært udtryk), og det betyder, at vi på dette sted skal indsætte en frase konstrueret ved hjælp af dette, andet diagram (eller regulært udtryk). Ting, der fremstår i afrundede kasser, er leksikalske symboler, som vi blot skriver ned. Vi har beskrevet en

sekvens hørende til den givne nonterminal, såfremt det lykkes at komme ud af diagrammet ved pilen i højre side af diagrammet.

Men lad os vende tilbage til diskussionen af sammenhængen mellem abstrakt og konkret syntaks. Den konkrete syntaks ovenfor er ikke særligt hensigtsmæssig, da den er tvetydig. Sekvensen »`1+2*3`« kan genereres på to forskellige måder, og identificerer således ikke entydigt et abstrakt syntakstræ. Derfor indføres en præcedens mellem reglerne for plus og gange for at opnå en entydig læsning. Almindeligvis tildeles gange en *højere præcedens* end plus, hvor denne præcedens skal forstås således, at gange knytter argumenterne hurtigere til sig end plus. I eksempler »`1+2*3`«, at gangetegnet knytter `2` og `3` til sig, hvorefter `+` kan få lov til at tage de to tilbage værende udtryk som sine argumenter, dvs. `1` og `2*3`.[8]

Ved at indføre en nye nonterminaler, kan vi udtrykke denne præcedens i et syntaksdiagram på følgende måde.



Men på denne måde har vi gjort det umuligt, at nedskrive sekvenser som svarer til et abstrakt syntakstræ for gangereglen, hvor et af argumenterne er et syntakstræ svarende til plusreglen. Tag som eksempel et gangeudtryk sammensat af `1` og `2+3` eller udtrykt som et abstrakt syntakstræ:

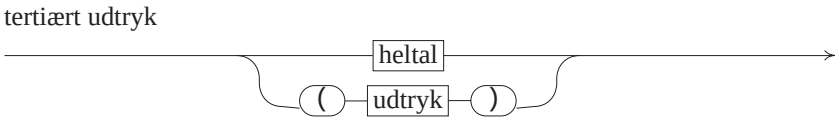
```
   *
  / \
 1   +
    / \
   2   3
```

Teksstrengen `1*2+3` vil uvægerligt blive forvekslet med et udtryk svarende til dette træ:

[8] Denne sprogbrug omkring præcedens er den etablerede, og men man skal være opmærksom på, at det vender modsat præcedenstallene, som kendes fra Prologs operatordefinitioner. Højt præcedenstal (i Prolog) svarer til lav præcedens i den klassiske sprogbrug og omvendt.

```
      +
     / \
    *   3
   / \
  1   2
```

Til dette formål udvider vi sproget med parantesudtryk, som vi kan indføje i diagrammet for tertiære udtryk som følger.



Når vi siger, at vi udvider sproget, er det vigtigt at fremhæve, at den abstrakte syntaks forbliver den samme, det er kun den konkrete syntaks, som udvides.

Man kan så gå videre med at opløse tvetydigheder ved indføre såkaldt associativitet af operatorerne. For en operator som minus er det essentielt med at præcisere associativiteten, da det (på det semantiske plan) er ret væsentligt, om »`1-2-3`« forstås på samme måde som »`(1-2)-3`« eller »`1-(2-3)`«. Almindeligvis opfattes »`(1-2)-3`« som det rigtige, svarende til at minus er *venstreassociativ*. Læse mere om præcedens og associativitet hos (Aho, Sethi, Ullman, 1986), som beskriver forskellige måde at indkooperere præcedens i grammatikken på.

En anden notationsform, som i udtrykskraft er ækvivalent med syntaksdiagrammer, er de velkendte BNF-diagrammer, som læseren forudsættes bekendt med. BNF står for Backus Normal Form eller Backus-Naur Form. Dog kan BNF ikke udtrykke løkker direkte, men det kan være nødvendigt at indføre flere nye nonterminaler. Syntaksdiagrammerne ovenfor kan omskrives til følgende BNF-regler:

⟨*udtryk*⟩ ::= ⟨*sekundært udtryk*⟩
| ⟨*udtryk*⟩ + ⟨*sekundært udtryk*⟩
⟨*sekundært udtryk*⟩ ::= ⟨*tertiært udtryk*⟩
| ⟨*sekundært udtryk*⟩ * ⟨*tertiært udtryk*⟩
⟨*tertiært udtryk*⟩ ::= ⟨*heltal*⟩
| (⟨*udtryk*⟩)

Man støder også på såkaldt EBNF (extended BNF), hvor notationen er udvidet operatorer for gentagelse m.v. svarende til de, vi benyttede i regulære udtryk.

Vi kan fremhæve en egenskab ved grammatiker, som er modsat tvetydighed, og den kaldes redundans. Grammatikken ovenfor med parentesudtryk er redundant fordi den samme abstrakte syntakstræ har flere konkrete fremtrædelsesformer. For eksempel repræsenterer

```
1+2*3,
1+(2*3),
(1)+2*3 og
((((1)+(2*3))))
```

det samme abstrakte syntakstræ.

Til slut beder vi læseren bemærke, at vi har angrebet sammenhængen mellem den konkrete, strukturelle syntaks og den abstrakte på uformel vis, men hvis grammatikkerne var tænkt som inddata til et metasprogligt system, f.eks. en oversættergenerator, så måtte der udvikles notationsformer herfor. Det forekommer ikke relevant, at opbygge særskilte datastrukturer til at repræsentere noget vi kunne kalde konkret-strukturelle syntakstræer. Når vi kommer til metoder til genkendelse af strukturel syntaks, vil vi i stedet konstruere abstrakte syntakstræer direkte — eller om muligt helt undgå at opbygge syntakstræer.

Beskrivelse af abstrakt syntaks

Abstrakt syntaks kan beskrives ved hjælp af BNF-notation eller syntaksdiagrammer, som vi anvendte tidligere for konkret, strukturel syntaks, men man er fri for at tænke i præcedens og associativitet og den slags. Den abstrakte grammatik for de tilbagevendende aritmetiske udtryk kan formuleres således:

$$
\begin{aligned}
\langle udtryk \rangle ::= \quad & \langle udtryk \rangle + \langle udtryk \rangle \\
| \quad & \langle udtryk \rangle * \langle udtryk \rangle \\
| \quad & \langle heltal \rangle
\end{aligned}
$$

Her skal man dog være klar over, at en BNF anvendt til abstrakt syntaks betyder noget andet end når den er anvendt til konkret syntaks. En »abstrakt« BNF definerer en mængde af abstrakte syntakstræer, og en »konkret« BNF en mængde af strenge af leksikalske symboler. Terminalsymbolerne i den abstrakte version tjener således blot til reference, og i princippet kunne man udelade dem, og blot navngive reglerne i stedet for.

$$
\begin{array}{lcll}
\langle udtryk \rangle & ::= & \langle udtryk \rangle \quad \langle udtryk \rangle & \text{(plus)} \\
& | & \langle udtryk \rangle \quad \langle udtryk \rangle & \text{(gange)} \\
& | & \langle heltal \rangle & \text{(tal)}
\end{array}
$$

12 Tilstandsmaskiner og leksikalsk analyse

Leksikalsk analyse refererer til de delprocesser i automatiske systemer til sprogbehandling, som handler om at opdele sekvenser af enkelte tegn (bogstaver, tal, blanktegn, osv.) i delsekvenser, som kan identificeres som repræsenterende leksikalske symboler. Tilstandsmaskiner ses som abstrakte beskrivelser af algoritmer, som drejer sig om analyse af tekststrenge eller, mere generelt, sekvenser af tegn over et vilkårligt alfabet. Her skal alfabet vel og mærket forstås i en bred betydning som en eller anden endelig mængde af simple symboler, som hver især ikke tilskrives nogen egentlig betydning, men hvor betydninger på et højere niveau er indkodet som (del-) sekvenser af disse symboler. Tilstandsmaskiner benyttes også i forbindelse med kontrolpaneler, som de findes på diverse elektroniske apparater, eller i forhold til grafiske og interaktive grænseflader, hvor »alfabetet« bl.a. indeholder museklik. Indenfor analyse af naturligt sprog benyttes tilstandsmaskiner også, men på en helt anden måde end det vi beskriver i denne bog. Overraskende nok kan hele analysen, incl. det vi almindelivis henfører til parsing ud fra på kontekstfri grammatikker, baseres på sammensætninger af mange forskellige tilstandsmaskiner; for en introduktion og yderligere referencer til disse meget spændende metoder, se f.eks. (Chanaud, 1999). Tilstandsmaskiner, eller alternativt regulære udtryk, benyttes ofte som specifikationssprog i såkaldt 4.-generationsværktøjer, oversættergeneratorer og andre systemer af en metasproglig art.

På den implementationsmæssige side skal det bemærkes, at visse lærebøger, f.eks. (Sedgewick, 1988) giver en noget misvisende behandling af emnet. Den refererede bog definerer tilstandsmaskiner på en ikke-standard måde og overser et af datalogiens klassiske resultater, nemlig at enhver ikke-deterministisk tilstandsmaskine på ganske enkel måde kan transformeres over i en deterministisk. Det giver således anledning til en implementation af tilstandsmaskiner baseret på en algoritme til at simulere generelle, ikke-deterministiske tilstandsmaskiner, som hverken er særligt effektiv eller særligt elegant.

I afsnit 4.2.2 introducerede vi regulære udtryk som et værktøj til at beskrive konkret syntaks, og tilstandsmaskiner, som introduceres i det følgende, er (abstrakte beskrivelser af) indretninger, som benyttes til at omforme en konkret tekst til sekvenser af leksikalske symboler. Tilstandsmaskiner kan indeholde nondeterminisme, og i afsnit 12.2 ser vi på, hvordan ethvert regulært udtryk kan omformes til en deterministisk tilstandsmaskine. Afsnit 12.3 beskriver to måder at implementere determiniske tilstandsmaskiner, den ene vha. en generel algoritme styret af en tabel, og den anden ved at producere en specialiseret algoritme svarende til hver tilstandsmaskine. Afsnit 12.4 benytter vi til at vise, hvordan tilstandsmaskiner kan bruges til at forklare strengsøgningsalgoritmer. I afsnit 12.4 udvider vi endelige tilstandsmaskiner med handlinger og viser, hvordan de to metoder for implementation kan generaliseres tilsvarende. Endelig, i afsnit 12.6, viser vi, hvordan leksikalsk analyse kan udføres af en passende tilstandsmaskine med tilknyttede handlinger.

Til den læser, som vil vide alt om tilstandsmaskiner og deres egenskaber, henvises til den dejlige lille bog af Hartmanis og Stearns (1966).

12.1 Tilstandsmaskiner

En tilstandsmaskine er en abstrakt maskine, som kan undersøge, om en givet tegnsekvens er omfattet af et givet regulært udtryk. Tilstandsmaskiner kaldes også ofte endelige automater. Vi starter med et definere fænomenet generelt.

En *tilstandsmaskine* består af

- en mængde af tilstande,
  - én af disse er en speciel *starttilstand*,
  - én eller flere af disse er specielle *sluttilstande*,
- en mængde af *transitioner*, som hver har
  - *begyndelses-* og *slutpunkt*, som er tilstande.
  - en *etikette*, som består af nul eller flere tegn.

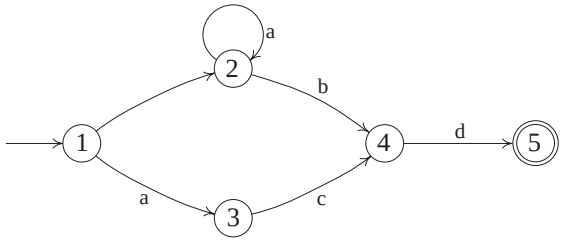
En given tilstandsmaskine beskrives nemmest grafisk ud fra følgende principper.

- tilstande tegnes som boller, evt. med et tal, som navngiver tilstanden.
- starttilstanden indikeres med en pil (uden startpunkt) pegende på sig,

– sluttilstande indikeres ved dobbelt optrukket boller.

– en transition tegnes som en pil mellem start og slutpunkt, den sidste angivet med en pilespids, og med påtegning af etiketten.

Et eksempel:



Denne tilstandsmaskine er istand til at genkende netop de strenge, som er beskrevet ved det regulære udtryk *(a*b + ac)d*. Vi kan definere dette begreb af genkendelse generelt.

Givet en tilstandsmaskine og en streng, så processeres strengen på følgende måde.

– Vi begynder i starttilstanden.

– Vi kan trække fra en tilstand til en anden, såfremt der er en pil imellem og

  – første tegn i strengen findes i etiketten; dette tegn fjernes fra strengen, eller

  – etiketten er tom; der sker ikke noget med strengen.

– Strengen er accepteret, såfremt vi på denne måde kan nå frem til en sluttilstand samtidigt med, at hele strengen er konsumeret.

For at acceptere strengen *aaabd* vil maskinen ovenfor gå gennem følgende tilstande.

Aktuel tilstand	aktuel streng
1	*aaabd*
2	*aaabd*
2	*aabd*
2	*abd*
2	*bd*
4	*d*
5	–

Denne maskine er et eksempel på en ikke-deterministisk maskine. Det skyldes, at når vi er i tilstand 1, og vi ved at første tegn i strengen er et *a*, så er det altså to muligheder. Vi kan gå til 3 mod at konsumere første tegn i strengen eller gå til 2 uden at ændre ved strengen. Definitionen af, hvad der kaldes deterministisk og hvad ikke, afspejler et princip om, at vi ønsker at implementere maskinen på en måde, hvor vi kun holder styr på én tilstand ad gangen, og hvor vi nøjes at kigge på første af de resterende tegn i strengen.

Definition. En tilstandsmaskine er *ikke-deterministisk*, såfremt den har en tilstand, hvor enten

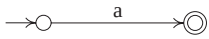
- samme tegn forekommer i etiketten på mere end én pil væk fra tilstanden, eller
- der går mere end én pil væk fra tilstanden, og en af dem har tom etikette.

En tilstandsmaskine, som ikke er ikke-deterministisk, kaldes *deterministisk*.

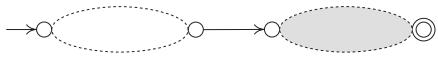
12.2 At lave regulære udtryk om til tilstandsmaskiner og at gøre disse deterministiske

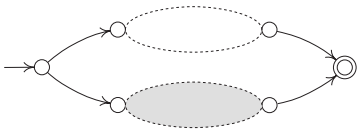
Regulære udtryk kan omskrives til tilstandsmaskiner på en enkel og systematisk måde. Dette kan beskrives som en rekursiv procedure, som går på strukturen af det regulære udtryk, således at hvert deludtryk svarer til sin maskine. Vi kan skitsere proceduren som følger. Konstruktionen foregår således, at hver tilstandsmaskine altid har netop en sluttilstand.

– ε svarer til 

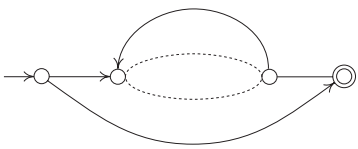
– et tegn *a* svarer til 

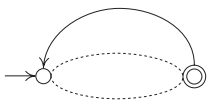
– hvis A svarer til  og B til 

så svarer AB til 

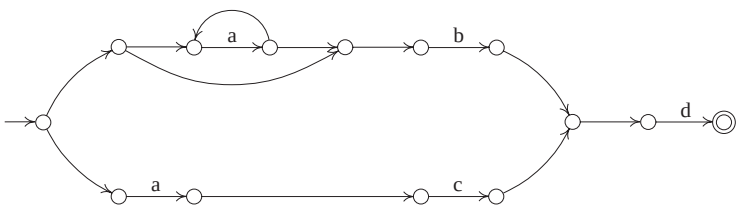
og A+B til 

– hvis A svarer til 

så svarer A* til 

og A^+ til 

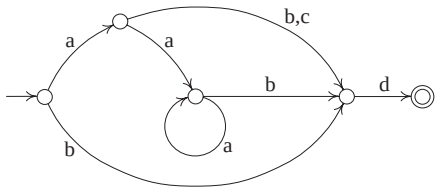
Dette var den generelle procedure, som virker i alle tilfælde, men hvis vi foretager det manuelt, kan vi gøre en del optimeringer ved at undlade at indføre nye knuder eller ved at slå knuder sammen. I afsnit 12.1 så vi en »manuelt fremstillet« maskine svarende til $(a^*b+ac)d$. Følger vi proceduren ovenfor når vi frem til følgende omstændige konstruktion.



Begrundelsen for indførelse af nye knuder og pile er at undgå problemer, hvis der er løkker tilbage til startknuden/slut-knuderne inde i de tilstandsmaskiner, som sættes sammen. Det kan man nemt overbevise sig om ved at lege lidt med at sammensætte diagrammer for a^* og b^* på forskellig vis.

På tilsvarende måde findes også en procedure, som forvandler en ikke-deterministisk maskine til en deterministisk. *Man kan således bevise, at enhver ikke-deterministisk maskine kan transformeres over i en deterministisk, som genkender de samme strenge*, se f.eks. Aho, Sethi, Ullman (1986), Aho, Ullman (1972), Sudkamp (1988). Denne procedure er lidt omstændig, men ikke særlig dybsindig. Alt i alt, så kan man sætte disse procedurer sammen og få et system ud af det som i den ene ende indlæser et regulært udtryk og producerer en deterministisk tilstandsmaskine som resultat. Typisk kobler man et sidste led på, så man får genereret en implementation af maskinen, sådan som vi vil beskrive det i afsnit 12.3.

Men det er faktisk ret nemt, manuelt at konstruere en deterministisk tilstandsmaskine direkte fra det regulære udtryk. Man starter med at tegne en starttilstand, og derefter udvider man diagrammet, så man får medtaget mere og mere af det regulære udtryk. Man skal blot sørge for ikke at komme til at tegne forgreninger, som vil indføre nondeterminisme. På denne måde kan man konstruere følgende deterministiske maskine[1] for vores favoritudtryk $(a^*b+ac)d$.



12.3 Implementation af deterministiske tilstandsmaskiner

Tilstandsmaskiner blev præsenteret som nogle abstrakte indretninger, men de, som er deterministiske, kan på nem måde skrives om til programmer i et traditionelt programmeringssprog.

Antag, vi har et programmeringssprog, som ligner Pascal, og at strengen læses tegn for tegn fra en fil. Hvis ch er en variabel at type »char«, så vil sætningen »read(ch)« lægge første tilbageværende tegn over ch og fjerne det fra input-filen. Yderligere antages (og sådan fungerer Pascal ikke, men det kunne man jo få det til), at der i slutningen af filen står et specielt tegn EOF. Og istedet for at gå ned, vil »read« blot fortsætte med at returnere EOF ved gentagne kald. Der er to forskellige måder at implementere en tilstandsmaskine på,

[1] Desværre er numrene faldet ud af tegningen af denne tilstandsmaskine; af hensyn til senere brug skal tilstandende nummereres fra venstre mod højre 1, 2, 3, 4 og 5.

– ved at indkode maskinen som en tabel, som så fortolkes af et program, som fungerer for alle deterministiske tilstandsmaskiner,

– ved at skrive en specialiseret algoritme, som i sit kontrolmønster simulerer maskinens tilstandsovergange.

De to metoder demonstreres for den deterministiske maskine i afsnit 12.2. Vi antager generelt, at alle programmer kan benytte sig af en global variabel ch og to labels, som svarer til, om det gik godt eller skidt. — Labels i Pascal er, som man måske husker, tal.

```
9999:   if ch = EOF then ♡♡♡♡ else goto 1313
1313:   † † † †
```

Der skal hoppes til label 9999, hvis maskinen har konsumeret en streng af den ønskede art, og testet er der for at kontrollere, at der ikke står noget sludder bagefter — i såfald må den samlede inputstreng forkastes. Label 1313 kan benyttes til fejlbehandling, f.eks. udsendelse af en fejlmeddelelse.

12.3.1 Implementation vha. tabel

Tilstandsmaskinen indkodes i følgende datastruktur,

```
var tabel: array[tilstand, char]
```

således, at når man står i en tilstand t og har læst tegnet x, så fås næste tilstand som tabel[t, x]. For vores eksempelmaskine side 182 ser tabellen sådan ud.

tilstand\ tegn	*a*	*b*	*c*	*d*	EOF	andre
1	2	4	-	-	-	-
2	3	4	4	-	-	-
3	3	4	-	-	-	-
4	-	-	-	5	-	-
5	-	-	-	-	*stop*	-

Ikke-udfyldte pladser, angivet ved »–«, svarer til en speciel værdi, vi kalder *fejl*. Antag også, der er en konstant *start*, som svarer til starttilstanden, i vores tilfælde tilstand 1. Den specielle værdi *stop* i tabellen fungerer som et stopkriterium. Den generelle fortolkeralgoritme bliver nu som følger.

```
t:= start;
repeat
    read(ch);
    t:= tabel[t, ch];
    if t = fejl then goto 1313
until t = stop ;
goto 9999
```

Denne algoritme, som fortolker en vilkårlig deterministisk tilstandsmaskine, er væsentlig enklerere og langt mere effektiv end en tilsvarende for potentielt ikke-deterministiske maskiner, jvf. (Sedgewick, 1988). Tidskompleksiteten svarer til længden af strengen, som analyseres. For hver tegn foretages én læseoperation, ét array-opslag samt to simple sammenligninger.

Man kunne måske indvende, at tabellen nemt kan blive meget stor, hvis der er mange tilstande, og man har én indgang for hvert eneste tegn i et normalt tegnsæt (f.eks. med 256 tegn). På den anden side, med nutidens RAM-priser…

12.3.2 Implementation vha. kontrolstrukturer

En anden og endnu mere effektiv metode er at oversætte tilstandsmaskinen direkte til programsætninger, således at hver tilstand svarer til en label i programmet. Ud for hver etikette læses så et tegn, og afhængigt af dette, hoppes til relevant etikette svarende til ny tilstand.

```
1: read(ch);
   case ch of
     'a': goto 2;
     'b': goto 4
   otherwise goto 1313;


2: read(ch);
   case ch of
     'a': goto 3;
     'b': goto 4;
     'c': goto 4
   otherwise goto 1313;


3: read(ch);
   case ch of
     'a': goto 3;
     'b': goto 4
   otherwise goto 1313;


4: read(ch);
   if ch = 'd' then goto 5 else goto 1313;


5: goto 9999;
```

Man bemærker algoritmens enkle form, som er opnået ved brug af den ellers så udskældte **goto**-sætning. Den enkle form gør også, at det vil være forholds-

vis nemt at skrive et program, som genererer algoritmen — ud fra et input, som beskrev en given tilstandsmaskine. Der findes programmeringssprog, eksempelvis Java, som ganske mangler **goto**, hvilket må anses for yderst uhensigtsmæssigt, hvis man ønsker at benytte programgeneratorer til at skabe Java-kildetekster, som det eksempelvis er tilfældet her. I Java må man så i stedet benytte en anden programstruktur, som antydes her i den Pascalagtige notation.

```
label:= 1;
repeat
    case label of
    1: begin read(ch);
          case ch of
            'a': label:= 2;
            'b': label:= 4
          otherwise label := 1313; end
    2: ...
    ...
until label = 1313 or label = 9999;
```

Altså, man må eksplicit programmere en løkke, som slår op i **case**-sætning styret af en variabel, vi hensigtsmæssigt kan kalde »label«, og **goto**-sætning simuleres ved at sætte en værdi til denne variabel.[2]

Man kan naturligvis også benytte programmeringssprogets løkke- og andre kontrolmekanismer. Man vil så naturligt udnytte analogien mellem de regulære udtryks »+« og en **repeat**-løkke, mellem »*« og **while**-løkken og så fremdeles.

12.4 Strengsøgning

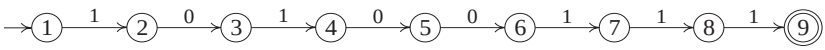
Vi viser her en generaliseret brug af tilstandsmaskiner, så de kan benyttes til at finde en tekststreng et arbitrært sted i en tekst. Den metode, vi beskriver her, svarer til den såkaldte Knuth-Morris-Pratt-algoritme; læs om baggrunden for den i (Sedgewick, 1988, kap. 19).

En tilstandsmaskine undersøger, som beskrevet ovenfor, hvorvidt en given streng er omfattet af et bestemt regulært udtryk. Strengsøgning kan formuleres på samme måde. Vi konstruerer blot en ny tilstandsmaskine, *som be-*

[2]Umiddelbart kunne det se ud som om man i et fattigt sprog uden **goto** ikke kan implementere tilstandsmaskiner effektivt. De teknologier, som anvendes i optimerende oversættere er faktisk så avancerede, at en god oversættere vil være istand til at opdage, at denne programstruktur kan implementeres effektivt ved hjælp af ubetingede hopinstruktioner. Hvorvidt det forholder sig således med de tilgængelige implementationer af Java vides ikke.

skriver samtlige tekster i hvilken, der forekommer en delstreng af den ønskede art.

Vi viser princippet vha. et eksempel. Antag, vi søger i strenge af 0'er og 1'er, og at den delstreng, vi leder efter, er 10100111. Følgende tilstandsmaskine vil acceptere netop denne streng og intet andet.

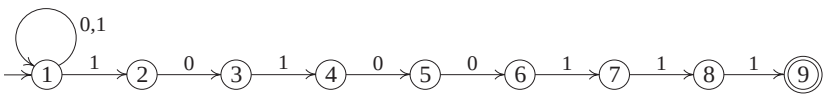


Vi kan f.eks. implementere denne maskine vha. »goto« som beskrevet i afsnit 12.3.2. Vi ændrer blot algoritmens afslutning således, at den ikke påstår fejl, hvis hele strengen ikke er konsumeret.

```
9999:  ♡♡♡♡
1313:  † † † †
```

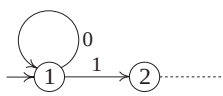
Det program, vi får ud af det, vil altså acceptere samtlige strenge, som begynder med 10100111. Så vidt så godt. Nu vil vi ændre på tilstandsmaskinen, så den også vil identificere søgestrengen, såfremt denne måtte optræde længere inde i teksten. Mere præcist formuleret, vi vil konstruere en tilstandsmaskine, som accepterer alle strenge, som begynder med arbitrært nonsens efterfulgt af det ønskede 10100111.

En måde at gøre dette på er at konstruere følgende ikke-deterministiske maskine.

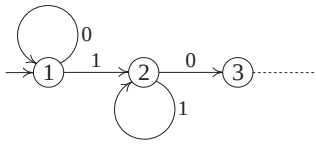


Den kan så køres igennem en algoritme af den slags, vi postulerede i afsnit 12.2, som fjerner ikke-determinisme. Den tilstandsmaskine, vi får ud af det, kan vi så igen køre igennem en generator, som producerer et program i stil med det i afsnit 12.2.2. Så er vi færdige, vi har et specialiseret og effektivt program til at søge efter den helt bestemte streng 10100111. Vi vil dog for princippets skyld vise, hvordan man manuelt kan konstruere denne ikke-deterministiske maskine.

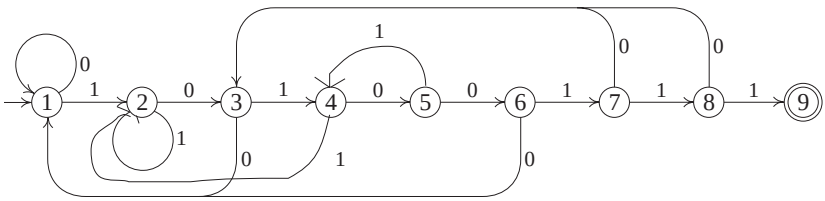
Hvis nu hele teksten begynder med et antal 0'er inden det første 1, så skal vi i alle tilfælde skippe forbi disse. Det gøres ved at tilføje følgende pil til den oprindelige tilstandsmaskine



Hvis nu maskinen har konsumeret nul eller flere 0'er, den møder 1 og derefter nok en 1, hvad så? Vor »hypotese« om, at det første 1 angav starten på vor søgestreng, var altså falsk. Men i stedet kunne det jo være, at det næste 1 angav starten. Derfor sætter vi en løkke tilstand 2 som følger.



Vi fortsætter ned igennem maskinen og sætter passende »bagud-pile« på, enten til starttilstanden eller en tilstand svarende til et punkt inde i strengen. Er vi eksempelvis nået til tilstand 8 og observerer et 0, så må vi konstatere, at de foregående læste 1010011 ikke er begyndelsen på en forekomst af søgestrengen. Men vi har altså læst 10100110, og hvis vi nu gik tilbage til starttilstanden, så ville vi jo overse den mulighed, at de sidste to læste tegn 10 kunne være begyndelsen af søgestrengen. Derfor går vi til tilstand 3, som netop repræsenterer, at der er læst 10. Tilsvarende transitionen fra tilstand 5 med tegnet 1 til tilstand 4, dette betyder, at der er læst 10101, og de understregede 101 kunne jo også være starten på søgestrengen; tilstand 4 svarer til at 101 er læst. Lignende forklaringer kan gives for transitionerne fra tilstande 7 og 8 til 3 og fra 6 til 2. I tilfældet, hvor vi i tilstand 6 møder tegnet 0, betyder det, at der er læst 000, hvilket ikke på nogen måde kan forekomme i søgestrengen, derfor tilbage til start. Summa summarum, så når vi frem til følgende, deterministiske maskine.



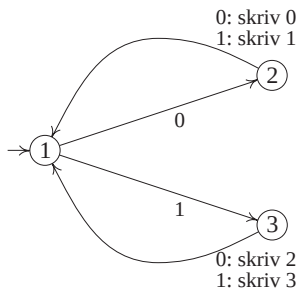
Den kan laves om til en tabel eller et program, som beskrevet i afsnit 12.3, og vi har løst vor opgave, nemlig at kunne lede efter strengen 10100111.

Metoden kan selvfølgelig generaliseres til at lede efter mønstre, som er beskrevet ved et arbitrært regulært udtryk. Analysen og tilpasningen af den tilhørende tilstandsmaskine bliver blot en anelse mere kompliceret. — Men man kan altid henvise til den automatiserede metode.

12.5 Handlinger

Som vi har beskrevet tilstandsmaskiner indtil nu, er de indretninger, som kan acceptere eller forkaste tekststrenge, hverken mere eller mindre. Ofte udvides tilstandsmaskiner med handlinger, således at hver overgang fra en tilstand til en anden kan udløse en handling. En handling kan bestå i modifikation af programvariable eller udsendelse af signaler til styring af forskelligt udstyr. Det sidste er relevant, hvis f.eks. tilstandsmaskinen skal fortolke input fra et kontrolpanel. Ved leksikalsk analyse, som vi ser på i afsnit 12.6, er det relevant at opsamle de læste tegn i en buffer med henblik på at kunne sige (f.eks.) hvilket variabelnavn, som er blevet genkendt.

Den grafiske notation for tilstandsmaskiner kan generaliseres med handlinger ved at notere dem ved pilene. Som eksempel kan vi tage et sprog af binære tal, som oversættes til 4-tals system. Vi forventer et de binære cifre altid kommer i et lige antal, og handlingerne består af udskrivningsordrer.



Overgangene fra tilstand 1 til 2 og fra tilstand 1 til 3 har tomme handlinger, overgangene fra 2 (og 3) har handlinger, som afhænger af, hvilket tegn, som gav anledning til tilstandsovergang. Hvis strengen 110100 processeres af denne tilstandsmaskine, vil de udførte skrivesætninger producere strengen 310.

Omend trivielt, så illustrerer dette eksempel begrebet »mode«.[3] Betydningen af, eller mere præcist, den handling som udløses af f.eks. tegnet 1, afhænger af forhistorien. Hvis forhistorien har ledt os til tilstand 2, vil læsning af tegnet 1 resultere i udskrift af »1«, og hvis forhistorien havde ledt os til tilstand 3, udskrives »3«.

Dette er helt analogt til de »modes«, der ofte er omkring betjeningspaneler. Tag f.eks. et tilfældigt stykke audioudstyr. Det har som regel en gevaldig masse knapper på forsiden, herunder et taltastatur. Hvilke knapper, man har

[3] Engelsk, udtales måud.

trykket på forinden, har stor indflydelse på, hvilken effekt man får ud af trykke 117.

Den generelle, tabelstyrede fortolkeralgoritme for deterministiske tilstandsmaskiner kan umiddelbart generaliseres til at håndtere handinger. Vi udvider med en ny tabel,

```
var handlings_tabel: array[tilstand, char] of handling
```

hvor så datatypen »handling« er en angivelse af de mulige handlinger, f.eks. ved et heltal. Algoritmen bliver som følger (vi minder om at »tabel« angiver transitioner).

```
t:= start;
repeat
   read(ch);
   h:= handlings_tabel[t, ch];
   t:= tabel[t, ch];
   if t = fejl then goto 1313 else udfør(h)
until t = stop ;
goto 9999
```

De faktiske handlinger kan placeres i en procedure på følgende måde.

```
procedure udfør(h: handling);
begin
   case h of
      1: <kode for handling 1>
      2: <kode for handling 2>
      ...
   end
end
```

Handlinger kan også nemt flettes ind i den implementation, vi viste i afsnit 12.3.2, hvor tilstandsmaskinen blev omskrevet direkte til et program. De programsætninger, som repræsenterer handlinger, skal så placeres umiddelbart foran de **goto**-sætninger, som repræsenterer tilstandsskift. Vor lille tilstandsmaskine ovenfor til to-tals-system-til-fire-tals-system-oversættelse, kommer til at se således ud.

```
1: read(ch);
   case ch of
    '0':  goto 2;
    '1':  goto 3
   otherwise goto 9999;

2: read(ch);
   case ch of
    '0':  write('0'); goto 1;
    '1':  write('1'); goto 1;
   otherwise goto 1313;

3: read(ch);
   case ch of
    '0': write('2'); goto 1;
    '1': write('3'); goto 1;
   otherwise goto 1313;
```

12.6 Leksikalsk analyse

Leksikalske analyse er relevant for alle sprog, som underkastes maskinel behandling. Som eksempler kan nævnes programmeringssprog eller naturligt sprog i et tekstbehandlingssystem med stave- eller syntakskontrol. Den leksikalske syntaks for et sprog beskriver, hvordan de enkelte tegn (bogstaver, cifre, punktuationer, osv.) skal opfattes som repræsenterende symboler på et højere abstraktionsniveau.

Betragt for eksempel følgende udsnit af et Pascal-program; blanktegn angives som »∆«, linieskift som »¶«.

```
∆∆begin¶∆∆∆∆x:=∆17
```

Der optræder i alt 18 tegn, men det er også korrekt at sige, at der optræder 4 leksikalske symboler, nemlig det specielle kodeord, som begynder sammensatte sætninger eller blokke, et variabelnavn, et symbol for assignmentsætning samt én talkonstant. Blanktegn og linieskift er i denne sammenhæng blot en måde at indikere start og slut på de enkelte symboler, men ellers er de ikke betydningsbærende i nogen forstand. Eksempelvis er det klart (ud fra den indoktrinering, vi har været udsat for i vor opvækst), at det ikke er tale om ét variabelnavn `beginx`, og heller ikke om to variablenavne `be` og `gin`.

Vi har tidligere vist, hvordan den leksikalske syntaks for et sprog kan beskrives ved

- til hvert leksikalsk symbol i sproget at angive, hvilke mulige sekvenser af tegn, som kan repræsentere det, og
- hvorledes en tegnsekvens skal opdeles i adskilte symboler.

Til at specificere det første anvendes ofte regulære udtryk eller tilstandsmaskiner. Til det sidste er der ikke nogen etableret standard. Vi kan dog henvise til systemet LEX, som har en generel notation for leksikalsk syntaks (se f.eks. Aho, Sethi, Ulman, 1986). LEX benyttes til at generere programmer, som leverer leksikalsk analyse til oversættergeneratoren YACC.

For at benytte tilstandsmaskiner og de implementationsteknikker, vi har udviklet i dette kapitel, skal vi altså

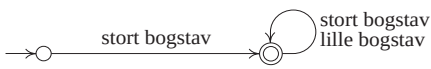
- danne en tilstandsmaskine, som beskriver alle tegnsekvenser, som er sammensat på korrekt måde af tegnsekvenser svarende til de mulige leksikalske symboler,
- tilføjer handlinger, som identificerer og signalerer, hvilke leksikalske symboler, som optræder.

Til det sidste er det ofte relevant også at opsamle den faktiske teksstreng, f.eks. svarende til variabelnavne. Vi benytter her den leksikalske syntaks for Datalog til at illustrere de generelle principper.

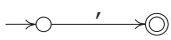
De leksikalske symboler for sproget Datalog, som vi beskrev i afsnit 4.3.1 vha. regulære udtryk, kan hver især omskrives til tilstandsmaskiner som følger.

Atomer:

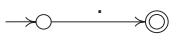
Variable:



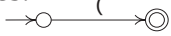
Komma:



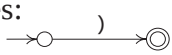
Punktum:



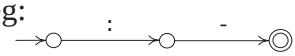
Startparentes:



Slutparentes:



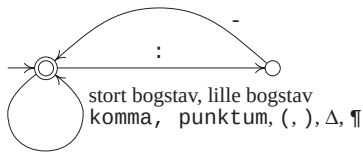
Kolon-streg:



Vi skal nu have flettet disse maskiner sammen til én maskine, som overholder tilføjer vi følgende konventioner, som også er en del af Datalogs leksikalske syntaks.

- Hvert atom og hver variabel udspænder en så stor delstreng som muligt,
- blanktegn og linieskift er tilladt mellem leksikalske symboler og ignoreres iøvrigt.

Lad os i første omgang overveje at konstruere en tilstandsmaskine, som acceptere tekststrenge, som er i overensstemmelse med den beskrevne leksikalske syntaks. Det vil sige, netop de teksstrenge, som er sammensat af nul eller flere (tekststrenge svarende til) leksikalske symboler, hvor der mellem hver af disse kan forefindes en sekvens på nul eller flere blanke eller linieskift. Men, som vi kan vise ved et eksempel, det er ikke et tilstrækkeligt krav til en tilstandsmaskinen.



Denne tilstandsmaskine genkender den relevante mængde af strenge, og den er deterministisk, men den er ikke særligt hensigtsmæssig i forhold til leksikalsk analyse, idet tilstandsovergangen ikke udtrykker noget om, hvilke symboler, som faktisk er genkendt. Med andre ord, der er ikke en indlysende måde at sætte relevante handlinger på.

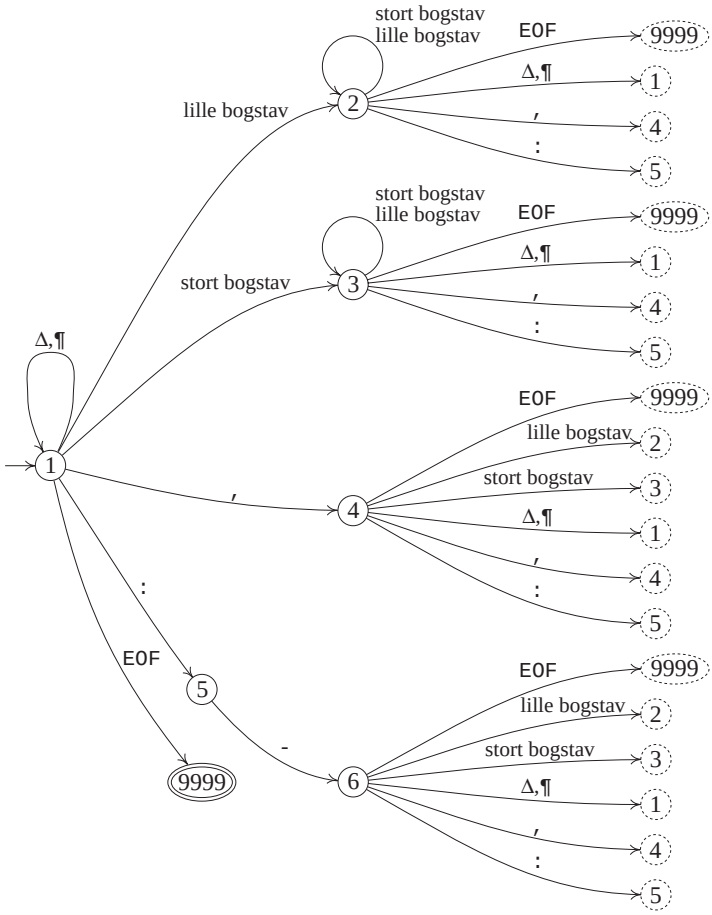
Vi vil i stedet bygge en maskine ved at sammensætte maskinerne for de enkelte symboler, således at

– når der optræder et tegn *t*, som signalerer at det »aktuelle symbol« er slut, trækkes over i et tilstand, som svarer et leksikalsk symbol, som kan begynde med *t*.

– Sekvenser af blanke og linieskift (og evt. andre tegn uden betydning) betragtes som et leksikalsk symbol; en tilstand svarende til genkendelse af sådanne sekvenser udnævnes til den samlede maskines starttilstand.

For at forenkle konstruktion af handlingstabellen antager vi yderligere, at enhver korrekt tekststreng slutter med det tegn symbol EOF. Herved opnås, at afslutningen af også det sidste leksikalske symbol registreres med en tilstandsovergang.

Her følger en sådan tilstandsmaskine svarende til Datalogs leksikalske syntaks, idet vi dog af hensyn til overskueligheden har set bort fra de leksikalske symboler svarende til parenteser og punktum. Når en tilstand er tegnet stiplet, skal den opfattes som en reference til en tilstand, som optræder andetsteds i diagrammet (dette er blot for at undgå for mange krydsende linier, der er ikke tale om en udvidelse af formalismen).



For at specificere relevante handlinger antager vi følgende faciliteter til rådighed.

- En buffer til opsamling af tegn for atomer og variable; til bufferen hører operationer til at initialisere og til at tilføje en enkelt tegn.
- Et ikke nærmere specificeret signalprimitiv, hvormed man kan sende beskeder til en anden (ikke nærmere specificeret) proces, som forventes at være interesseret i de leksikalske symboler.

Handlinger til tilstandsmaskiner ovenfor udformes som følger.

- Til enhver overgang fra tilstand 2 til tilstand 2:

  Tilføj læste tegn til bufferen.

- Til enhver anden overgang, som ender i tilstand 2:

Initialiser buffer og tilføj det læste tegn.

– Til enhver anden overgang, som udgår fra tilstand 2:

  Send et signal om, at et atom er genkendt og inkluder det aktuelle bufferindhold.

– Helt analogt vdr. overgange til og fra tilstand 3; her signaleres dog, at en variabel er genkendt til sidst.

– Til enhver anden overgang, som udgår fra tilstand 4:

  Send et signal om, at et komma er genkendt

– Til enhver anden overgang, som udgår fra tilstand 6:

  Send et signal om, at kolon-streg er genkendt.

– Alle andre handlinger er tomme.

Bemærk, at denne specifikation i visse tilfælde angiver to handlinger og i såfald udføres de begge. (F.eks. fra 6 til 2, skal signaleres, at komma er genkendt samtidigt med at bufferen initialiseres og tilføjes det læste tegn).

Vedrørende implementation, så kan såvel tabelmetoden, som opskrivningen til kontrolstruktur benyttes, jvf. afsnit 12.5. Kommunikation med en proces, som modtager meddelserne (typisk en parser, som »forbruger« leksikalske symboler) kan foregå på flere måder.

– Ved corutiner, så tilstandsmaskinen lægger sig til at sove (i Simula: »detach«) efter hver udsendelse af signal, og modtagerprocessen genstarter den (i Simula: »resume«) hver gang, der er brug for en ny meddelelse.

– Vha. »streams« eller »pipes«, hvor processerne programmeres hver især sekventielt, og det underliggende operativsystem sørger for en udførelse, som svarer til, hvad der kan opnås manuelt med corutiner.[4]

– Algoritmen, som simulerer tilstandsmaskinen, opdeles i en initialiseringsdel samt en procedure, som hver gang den kaldes, foretager så stor del af processen, at en meddelelse kan produceres. Vi viser en sådan algoritme i det følgende.

[4]Dette er nært beslægtet med den teknik, som kaldes »lazy evaluation«.

12.7 Eksempel: Leksikalsk analyse for Datalog i Simula

Vi viser her en implementation af leksikalsk analyse for Datalog, hvor det vigtige er at vise, hvordan de relevante programdele kan pakkes ind i klasser og procedurer, så de kan integreres i et større program.[5]

Den leksikalske analyse kan indkapsles i en klasse, `LEKSIKALSK_SCANNER`, der som parameter har navnet på den fil, der skal læses fra, og som attribut har den en procedure, som returnerer symbolerne ét efter ét.

Af hensyn til vores parsemetode vil vi yderligere udstyre klassen med en mulighed for at undersøge arten af det næste symbol uden at fjerne det. Hvis parseren for eksempel er igang med at læse argumenterne i et mål, og den netop har læst et komma — hvis det næste symbol er `ATOM_LEX`, ja, så skal den kalde proceduren for genkendelse af atomer, og hvis det er en `VARIABEL_LEX`, kaldes en procedure, der genkender variable. Tilsvarende, hvis parseren netop har læst en klausul i et program og det næste symbol er `ATOM_LEX`, så skal den kalde en procedure, der indlæser en klausul, og denne procedure skal naturligvis have mulighed for at læse dette første prædikatsymbol i klausul. — Mere om parsing i kapitel 13.

Vi får da følgende klasseskelet:

```
class LEKSIKALSK_SCANNER(file_name);
  value file_name; text file_name;
begin
  ref(LEKSIKALSK_SYMBOL) look_ahead;
  ref(LEKSIKALSK_SYMBOL) procedure read_symbol; . . .
  . . .
end;
```

Når der oprettes et objekt af klassen `LEKSIKALSK_SCANNER` skal filen åbnes, og det første leksikalske symbol skal indlæses og placeres i attributten `look_ahead`. Så er alt sat op til parseren, den kan se det første symbol på filen i attributten `look_ahead` og konsumere dette og de efterfølgende vha. `read_symbol`. Proceduren `read_symbol` skal returnere den gældende værdi af `look_ahead` og sætte `look_ahead` til det næste symbol, hvis tegnsekvens indlæses fra filen.

Vi kan altså uddybe beskrivelsen af `read_symbol` og klassens initialiseringsdel som følger:

[5] Beklageligvis implementerer den viste algoritme ikke eksakt tilstandsmaskinen vist i det foregående. Den anvendte algoritme er fra en tidligere version af kursusnoter, og i en fremtidig version vil den erstattes af en algoritme konstruktureret systematisk med **goto**, som er beskrevet tidligere.

```
class LEKSIKALSK_SCANNER(file_name);
  value file_name; text file_name;
begin
  ref(LEKSIKALSK_SYMBOL) look_ahead;
  ref(LEKSIKALSK_SYMBOL) procedure read_symbol;
  begin
    read_symbol:- look_ahead;
    look_ahead:-det næste symbol;
  end;
  . . .
  read_symbol;
end;
```

Kaldet af `read_symbol` placerer første leksikalske symbol i `look_ahead`, dvs. initialiserer denne attribut. (Egentligt er `read_symbol` en procedure-funktion. Første kald af den returnerer ingen meningsfyldt værdi (`none`, faktisk), det interessante er her dens sideeffekt!)

Der er her benyttet kontrolstrukturer i `read_symbol` for at opløse filens sekvens af tegn til leksikalske symboler, men hvor det havde været mere elegant at anvende en af en tabel eller »go to«- metoden.

Hvis nu den leksikalske analyse står over for at skulle læse et nyt leksikalsk symbol, og det første tegn, den ser, er et stort bogstav, ja, så er det klart, at der er tale om en variabel. Gennemgår vi beskrivelsen af Datalogs leksikalske syntaks, viser dette sig at være et generelt fænomen: Det første tegn i et leksikalsk symbol bestemmer entydigt, hvilken slags symbol, der er tale om. Når det leksikalske symbol således er klassificeret, fortsætter vi med at læse tegn, sålænge det allerede indlæste er i overensstemmelse med det relevante syntaksdiagram. For en variabel, for eksempel, læses videre sålænge, der er bogstaver at læse fra filen.

Også på tegnniveau bruger vi fidusen med at kigge et emne længere frem; hvis i tilfældet med en variabel, at indlæsningen af dens navn standses, fordi der mødes en slutparentes, så skal denne slutparentes jo gemmes til næste leksikalske symbol. Til dette formål indfører vi to lokale attributter, en variabel `character_look_ahead` og en procedure-funktion `character_read`, der fungerer fuldstændigt analogt til de tilsvarende for leksikalske symboler.

Klassedefinitionen er gengivet med sin fulde tekst nedenfor. I forhold til diskussionen ovenfor er der blot tilføjet nogle detaljer for at fjerne eventuelle blanktegn mellem leksikalske symboler og til at sikre, at der ikke forsøges læst forbi filens afslutning. Læseren opfordres til omhyggeligt at kontrollere, at koden er korrekt, evt. med lidt hjælp af det efterfølgende eksempel. Proceduren `error` skriver ganske enkelt en fejlmeddelelse ud og standser pro-

gramudførelsen; vi vil senere diskutere mere brugervenlige former for fejlbehandling.

```
class LEKSIKALSK_SCANNER(file_name);
  value file_name; text file_name;


begin
  ref(LEKSIKALSK_SYMBOL) look_ahead;
  character character_look_ahead;


  ref(infile) file;


  character procedure character_read;
  begin
    character_read:= character_look_ahead;
    if not file.endfile then
      character_look_ahead:= file.inchar
  end character_read;


  ref(LEKSIKALSK_SYMBOL) procedure read_symbol;
  begin
    read_symbol:- look_ahead;
    ! set look_ahead to the next LEKSIKALSK_SYMBOL;
    while character_look_ahead = ' '
              and not file.endfile do
      character_read;
    if file.endfile then
      look_ahead:- new END_OF_FILE
    else if 'a' <= character_look_ahead and
             character_look_ahead <= 'å' then
      begin
        text string; string:- blanks(100);
        while letter(character_look_ahead) do
          string.putchar(character_read);
        look_ahead:- new ATOM_LEX(string.strip)
      end
    else if 'A' <= character_look_ahead and
            character_look_ahead <= 'Å' then
      begin
        text string; string:- blanks(100);
        while letter(character_look_ahead) do
          string.putchar(character_read);
        look_ahead:- new VARIABEL_LEX(string.strip)
      end
    else if character_look_ahead = ',' then
      begin
```

```
        look_ahead:- new KOMMA;
        character_read
      end
    else if character_look_ahead = '.' then
      begin
        look_ahead:- new PUNKTUM;
        character_read
      end
    else if character_look_ahead = '(' then
      begin
        look_ahead:- new START_PARENTES;
        character_read
      end
    else if character_look_ahead = ')' then
      begin
        look_ahead:- new SLUT_PARENTES;
        character_read
      end
    else if character_look_ahead = ':' then
      begin
        character_read;
        if character_look_ahead = '-' then
          begin
            character_read;
            look_ahead:- new KOLON_STREG
          end
        else
          error("- FORVENTET EFTER :")
      end
    else error("ULOVLIGT TEGN")
  end read_symbol;


  ! Initialize file:  ;
  file:- new infile(file_name);
  file.open(blanks(100));


  ! Initialize look_ahead:  ;
  character_read;
  read_symbol
end LEKSIKALSK_SCANNER;
```

Et lille eksempel kan tjene til at illustrere samspillet mellem de forskellige lag. Antag, at der oprettes et objekt af klasse `LEKSIKALSK_SCANNER` for en fil med følgende tekst.

```
pred( atom, Var ...
```

I klassens initialiseringsdel åbnes filen og proceduren `character_read` kal-

des én gang for at få initialiseret `character_look_ahead`. De interessante variable har da følgende værdier:

```
look_ahead:  none
character_look_ahead:  'p'
tilbage at læse på filen:  red( atom, Mar ...
```

Nu er forudsætningen for at `read_symbol` kan kaldes til initialisering af `look_ahead` opfyldt. Vi fortsætter nu initialiseringen ved at kalde denne procedure; `read_symbol` vil da kalde `character_look_ahead` fire gange, hvorefter vi har dette billede:

```
look_ahead:  ATOM_LEX("pred")
character_look_ahead:  '('
tilbage at læse på filen:  atom, Mar ...
```

Nu er forudsætningen for at parseren kan begynde sit arbejde opfyldt: Den kan se, at det første symbol på filen er et atom, og efter at den har kaldt `read_symbol`, bliver billedet som følger:

```
look_ahead:  START_PARENTES
character_look_ahead:  ' '
tilbage at læse på filen:  atom, Mar ...
```

Ved efterfølgende kald af `read_symbol` vil filens leksikalske symboler optræde et efter et i `look_ahead`. Tilsidst, når filen er læst til enden, indeholder `look_ahead` det specielle leksikalske symbol `END_OF_FILE`, hvorefter det ikke mere er meningsfyldt at kalde `read_symbol`.

12.8 Afsluttende diskussion af leksikalsk analyse

Vi har skitseret et generel metode ved hjælp af et eksempel, og vi vil her diskuterer generaliteten.

Det var stiltiende forudsat, at tilstandsmaskinen, som kom ud af at sætte de mange »små« maskiner sammen, er deterministisk. Dette indebærer en forudsætning om, at hvert leksikalske symbol kan identificeres på sit første tegn (helt analog til betingelsen i forbindelse med rekursivt nedstigende parsing).

Den betingelse holder desværre sjældent. Tænk på et traditionelt sprog som Java, hvor der er reserverede ord som »begin« og identifiere som »beginning«. I dette tilfælde vil man typisk i tiltandsmaskinen slå reserverede ord og identifiere sammen, og så lade den afsluttende handling ved et tabelopslag afsløre det læste symbols klasse.

Som konklusion må vi konstatere, at tilstandsmaskiner er et nyttigt element i en teoretisk forståelse af leksikalsk analyse og som udgangspunkt for

at konstruerer effektive implementationer. Til gengæld må der stilles spørgsmålstegn ved, om den manuelle konstruktion, vi antydede i eksemplet, kan siges at være en velegnet metodik til konstruktion af programmer til leksikalsk analyse. Selv i dette meget simple eksempel var der besvær med overhovedet at tegne den resulterende maskine, og man kan meget nemt begå en fejl i en eller anden tilstandsovergang eller handling.

Det må anbefales, at man benytter sig af automatiske metoder. Dvs. man formulerer en beskrivelse af en leksikalsk syntaks i et passende metasprog, og når ellers værktøjet har godkendt beskrivelsen som havende passende egenskaber vdr. determinisme, så konstrueres det endelige program automatisk. — LEX, som er omtalt ovenfor, fungerer på denne måde.

Leksikalsk analyse er ansvarlig for en meget stor andel af tidsforbruget for en compiler, og derfor er det oplagt at undersøge muligheden for uderligere optimeringer. Vi kan her referer til (Aho, Sethi, Ullman, 1986), som beskriver, hvordan man kan optimere ved at undlade brug af højniveaulæseprocedurer for de enkelte tegn og i stedet operere direkte på de buffervariable, som operativsystemet benytter i forbindelse med overførsel af information fra harddisk til RAM.

12.9 Opgaver

Opgave 12.1 Skriv et regulært udtryk, som repræsenterer tal. Formatet er illustreret ved flg. eksempler.

1356	
– 3.17	
35.001	
7e6	(syv millioner)
6.9997e6	(ca. syv millioner)

Tegn en tilsvarende, deterministisk tilstandsmaskine. Konstruér en tilstandstabel svarende til den i afsnit 12.3.1 og en algoritme svarende til den i afsnit 12.3.2. Foretag endelig en håndsimulering af de to algoritmer, når de udsættes de for eksempelstrenge som er vist ovenfor.

Opgave 12.2 Konstruktionen af et program i afsnit 12.3.2 udfra en deterministisk tilstandsmaskine er beskrevet ud fra et eksempel. Desværre indeholder eksemplet ikke tomme transitioner, dvs. transitioner, som ikke konsumerer et tegn. Giv et eksempel på en tilstandsmaskine, som indeholder tomme transitioner og stadig er deterministisk og forklar, hvordan metoden i afsnit 12.3.2 kan generaliseres til at håndtere dette.

Opgave 12.3 Konstruér en tilstandsmaskine, som accepterer strenge af 0'er og 1'er, i hvilken delstrengen

```
100110011001
```

forekommer, jvf. afsnit 12.5.

Opgave 12.4 I afsnit 12.3 forklarede vi to måder at implementere deterministiske tilstandsmaskiner på, ved en tabel, som kunne fortolkes af en generel algoritme, eller ved et specialiseret (og mere effektivt) program. Opstil forudsætninger, som afgør, hvornår den ene og den anden metode er at foretrække. Giv eksempler på anvendelser, som illustrerer begge tilfælde.

Opgave 12.5 I en ikke nærmere angivet republik er telefonnumrene på 5 cifre. Ambassaden for et naboland, som ikke er særlig velset, har følgende telefonnummer.

```
7 9 5 7 4
```

Efterretningstjenesten i republikken aflytter en bestemt slags meddelelser indkodet som sekvenser af decimale cifre, og ønsker at identificere de meddelelser, hvori det nævnte telefonnummer forekommer. Tegn en *deterministisk* tilstandsmaskine, som genkender netop de meddelelser, som indeholder det odiøse telefonnummer.

Opgave 12.6 I afsnit 12.6 antydede vi en tilstandsmaskine med handlinger, som kunne kunne foretage leksikalsk analyse for en delmængde af Datalogs syntaks. Vi skitserede yderligere, hvordan denne tilstandsmaskine kunne omskrives til en algoritme (efter **goto**-princippet). Skriv denne algoritme under antagelse at et passende signaleringsprimitiv er tilgængeligt.

Opgave 12.7 Opskriv komplette transitions- og handlingstabeller for tilstandsmaskinen omtalt i opgave 12.6.

13 Genkendelse af strukturel syntaks, også kaldet parsing

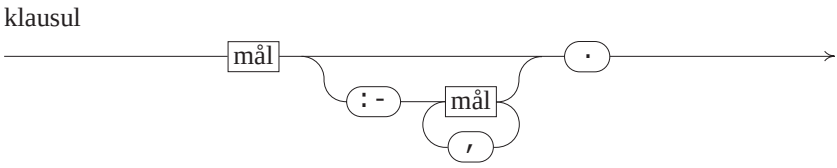
Når de leksikalske symboler i en tekst er identificeret, giver det mening at forsøge at identificere den strukturelle syntaks for derved at kunne rekonstruere de abstrakte syntakstræer, som måtte have eksisteret i hovedet på den programmør eller bruger, som har indtastet den tekst, der er under analyse.

I klassiske formuleringer af teorien om syntaksgenkendelse, har man benyttet stakmaskiner (på engelsk: push-down automata), som er generaliseringer af endelige tilstandsmaskiner med en stak. Vi viser her to principper for genkendelse af strukturel syntaks, top-down repræsenteret ved recursive-descent metoden, hvor stakken camoufleres bag systemer af gensidigt rekursive procedurer, og bottom-up, som nemmest beskrives ved en eksplicit stak.

13.1 Rekursivt nedstigende parsing

Denne parsemetode udmærker sig ved, at de algoritmer, som udfører genkendelsen af strukturel syntaks er mere eller mindre umiddelbare omskrivninger af syntaksreglerne til kontrolstrukturer, og den gensidigt rekursive reference mellem diagrammer omskrives meget naturligt til rekursive procedurekald. På denne måde er rekursivt nedstigende parsing den mest elegante, og velegnet i undervisning om sprog, idet de strukturelle egenskaber af sprogene understreges.

Vi vil her beskrive metoden gennem et eksempel, en parser for Datalog. Lad os betragte følgende diagram for en klausul:



Når parseren på et givet tidspunkt skal læse en klausul, må den først indlæse et mål, nemlig klausulens hovede. Herefter er der to valgmuligheder, enten er der tale om et faktum — eller en regel med »`:-`« og en efterfølgende krop. Hvordan kan parseren nu finde ud af, hvilken vej, den skal vælge? Kigger vi lidt nærmere på syntaksdiagrammet, fremgår det, at hvis det næste, ulæste symbol er et punktum, så er det slut med reglen, altså et faktum. Er symbolet derimod »`:-`«, må vi vælge den anden vej gennem diagrammet, dvs. der skal indlæses en krop.

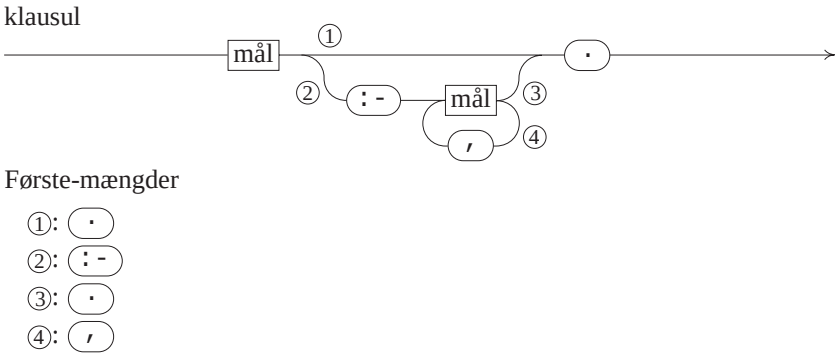
Vi skitserer nu den del af parserens algoritme, som bruges til at indlæse klausuler som følger, idet vi antager at variablen `source` referer til et objekt af klassen `LEXICAL_SCANNER`. Vi benytter her Simula som beskrivelsessprog, men læsere, som er fortrolige med Java, skulle også kunne gennemskue teksten; se evt. appendiks B for en fuld kildetekst af Datalogsystemet skret i Java.

```
indlæs et mål;
inspect source.look_ahead
  when KOLON_STREG do
    begin
      source.read_symbol; !dvs. skip forbi :-;
      indlæs en liste af mål; !dvs. indlæs kroppen;
    end
  when PUNKTUM do !ingenting;
otherwise der er tale om en fejl;
if source.look_ahead is PUNKTUM then
  source.read_symbol !reglen slutter som den skal;
else
  der er tale om en fejl;
```

Vi ser altså, at algoritmen for indlæsning af klausuler er en simpel omformulering af det tilsvarende syntaksdiagram — og hvor der skal foretages et valg mellem et faktum eller en regel, eller mellem om der er flere mål eller ikke i en regels krop, afgøres dette ved at kigge på `look_ahead`.

Vi skal nu se lidt mere systematisk på, hvilke betingelser, en grammatik skal opfylde, for at den rekursivt nedstigende parse-metode kan anvendes. Kernen i metoden er, at parseren ved at kigge ét leksikalsk symbol frem kan navigere gennem syntaksdiagrammernes labyrint af alternative forgreninger. For hver sådan alternativ vej i et diagram, defineres dens *førstemængde* til at være de leksikalske symboler, som kunne være starten på en sekvens, der kan beskrives ad den pågældende vej. Vi kan notere førstemængderne ved at sætte etiketter på forgreningerne i et syntaksdiagram og for hver etikette angive de tilhørende førstemængder. Førstemængderne i diagrammet for klausuler, for

eksempel, kan markeres som følger.[1]



Omskrivningen af et syntaksdiagram til en rekursivt nedstigende parse-procedure fungerer hvis og kun hvis: For ethvert forgreningspunkt udi alternativer må intet symbol optræde i mere end en førstemængde. (Af årsager, vi ikke skal komme ind på her, kaldes denne egenskab for LL(1), jvf. Aho, Sethi, Ullman, 1986). I ovenstående eksempel ser vi, at diagrammet for klausuler overholder betingelsen, da mængde 1 og 2 ikke overlapper hinanden og 3 og 4 ikke overlapper hinanden.

Den interne repræsentation af det abstrakte syntakstræ for et Datalogprogram er defineret ved et antal klasser, en for hver syntaktiske kategori og for lister af mål og af argumenter, jvf. afsnit 4.3.2. Til hver sådan klasse knytter vi nu en indlæseprocedure. I et objektorienteret programmeringssprog betyder det, at vi knytter en virtuel procedure til overklassen `KATEGORI`, og så for hver kategori beskriver proceduren i detaljer.

Da formålet med indlæsningen er at opnå en intern repræsentation af et Datalogprogram, lader vi indlæseprocedurerne konstruere et træ svarende til den tekst, de hver især har læst. Vi får da følgende programstruktur:

[1]Eksemplet her er lidt uheldigt fra et pædagogisk synspunkt, da samtlige førstemængder består af netop et element. Der vil ofte være et større antal symboler i hver mængde, som således kan betinge, hvilken vej, parseren må vælge.

```
class KATEGORI;
  virtual: procedure read;
begin end;

KATEGORI class PROGRAM;
begin
  ...
  procedure read(source);
    ref(LEXICAL_SCANNER) source;
    ...
  end;

KATEGORI class KLAUSUL;
  begin
    ...
    procedure read(source);
      ref(LEXICAL_SCANNER) source;
      ...
  end;
```

osv.

Indlæsning af en konstruktion foregår således ved først at skabe et objekt af passende klasse og derefter aktivere dets indlæseprocedure. For hvert underobjekt (≈ undertræ ≈ delfrase) gentages processen; deraf metodens betegnelse, »recursive descent parsing«.

Vi bringer her de reviderede klassedefinitioner for klausuler og lister af mål med indlæseprocedurerne i fuld tekst; indlæseprocedurerne for de øvrige klasser går efter helt samme melodi (gengivet i appendices A og B).

```
KATEGORI class KLAUSUL;
begin
  ref(MÅL) head;
  ref(MÅL_LISTE) body;
     ! Empty body signifies a fact;

  procedure read(source);
    ref(LEXICAL_SCANNER) source;
  begin
    head:- new MÅL; head.read(source);
    inspect source.look_ahead
      when KOLON_STREG do
        begin
          source.read_symbol;
          body:- new MÅL_LISTE;
          body.read(source)
        end
      when PUNKTUM do ! nothing;
    otherwise error(":- eller . forventet");
    if source.look_ahead is PUNKTUM then
      source.read_symbol
    else
      error(". forventet")
  end read;
end KLAUSUL;

KATEGORI class MÅL_LISTE;
begin
  ref(GOAL) first;
  ref(MÅL_LISTE) rest;

  procedure read(source);
    ref(LEXICAL_SCANNER) source;
  begin
    first:- new MÅL;
    first.read(source);
    if source.look_ahead is KOMMA then
      begin
        source.read_symbol;
        rest:- new MÅL_LISTE;
        rest.read(source)
      end
  end read;
end MÅL_LISTE;
```

I 13.3 ser vi nærmere på fejlbehandling; den opmærksomme læser har måske allerede bemærket, hvordan viden om førstemængderne allerede er udnyttet i fejlmeddelelserne ovenfor.

13.2 Syntaksstyret oversættelse

Parseren præsenteret i det foregående afsnit er et eksempel på en *syntaksstyret* algoritme eller proces. Til hver regel i sprogets grammatik svarer en procedure, som, når den kaldes for et givet syntakstræ, rekursivt kalder procedurerne hørende til dens undertræer. Det generelle mønster for en syntaksstyret algoritme kan beskrives vha. virtuelle procedurer i Simula.

```
class KATEGORI;
  virtual procedure P;
begin end


KATEGORI class KAT1;
begin
  ref(KAT11) sub1;
  ref(KAT12) sub2;
  ...
  procedure P;
  begin
    ...
    sub1.P;
    ...
    sub2.P;
    ...
  end
end KAT1;


```

osv.

Mange af de processer, som optræder i forbindelse med sprog og syntakstræer, er af natur syntaksstyrede. Formatteret udskrift af programtekst er et eksempel: Hvordan udskriver man et Datalogprogram? Ved at udskrive de klausuler, programmet består af. Og hvordan udskriver man en klausul? Ja, man kan f.eks. starte på en tom linie, udskrive klausulens hovede, og hvis klausulen er en regel, så udskrives »:-«, hvorefter kroppen udskrives, sluttelig skrives et punktum, og reglen er færdigskrevet. Og på tilsvarende vis er udskriften af alle øvrige Datalogkonstruktioner bestemt ved en sammenstilling af udskrifterne af deres respektive delkonstruktioner.

Her følger Simula-teksten for en algoritme til acceptabel, formatteret udskrift af Datalogprogrammer.

```
class KATEGORI;
  virtual: procedure pretty_print;
begin
end KATEGORI;


KATEGORI class PROGRAM;
begin
  ref(KLAUSUL) først;
  ref(PROGRAM) rest;


  procedure pretty_print;
  begin
    first.pretty_print;
    if rest =/= none then
      rest.pretty_print
  end pretty_print;
end PROGRAM;


KATEGORI class KLAUSUL;
begin
  ref(MÅL) head;
  ref(MÅL_LIST) body;


  procedure pretty_print;
  begin
    outimage;
    head.pretty_print;
    if body =/= none then
       begin outtext(":-"); body.pretty_print end;
    outtext(".")
  end pretty_print;
end KLAUSUL;


KATEGORI class MÅL;
begin
  ref(ATOM) prædikat;
  ref(ARGUMENT_LISTE) argumenter;


  procedure pretty_print;
  begin
    predicate.pretty_print;
```

```
    outtext("(");
    arguments.pretty_print;
    outtext(")")
  end pretty_print;
end GOAL;


KATEGORI class MÅL_LISTE;
begin
  ref(MÅL) first;
  ref(MÅL_LISTE) rest;


  procedure pretty_print;
  begin
    outimage;
    outtext(blanks(4));
    first.pretty_print;
    if rest =/= none then
      begin
        outtext(",");
        rest.pretty_print
      end
  end pretty_print;
end GOAL_LIST;


KATEGORI class ARGUMENT;
begin
end ARGUMENT;


KATEGORI class ARGUMENT_LISTE;
begin
  ref(ARGUMENT) først;
  ref(ARGUMENT_LISTE) rest;


  procedure pretty_print;
  begin
    first.pretty_print;
    if rest =/= none then
      begin
        outtext(", ");
        rest.pretty_print
      end
  end pretty_print;
end ARGUMENT_LISTE;
```

```
ARGUMENT class ATOM;
begin
  text id;
  procedure pretty_print;
    outtext(id);
end ATOM;


ARGUMENT class VARIABEL;
begin
  text id;
  procedure pretty_print;
    outtext(id);
end VARIABEL;
```

En oversættelse fra et sprog til et andet, for eksempel fra et højniveausprog til maskinsprog, kan også være en syntaksstyret proces: Det, at oversætte en sammensat konstruktion, gøres ved at oversætte dens delkonstruktioner og sætte deloversættelserne sammen med passende klister. Tag for eksempel den velkendte tildelingssætning fra et traditionelt, imperativt programmeringssprog.

variabel := *udtryk*

Hvis nu variable altid oversættes til kode, som lægger en lageradresse i et bestemt register, *A*, og udtrykket til kode, der lægger en værdi i et andet register, *V*, da kan en vilkårlig tildelingssætning oversættes som følger:

- Oversæt variablen.
- Oversæt udtrykket.
- Genérer en instruktion, som flytter indeholdet af register *V* over i den lagercelle, hvis adresse ligger i *A*.

I dette eksempel forholdt det sig således, at de to deltræer skulle oversættes i den samme rækkefølge, som de er blevet indlæst. Hvis dette gælder for alle syntaksdiagrammer med tilhørende oversættelse, taler vi om en *simpel*, syntaksstyret oversættelse. Simpel, syntaksstyret oversættelse udmærker sig ved, at indlæsning og oversættelse kan flettes, så konstruktion af syntakstræer helt kan undgås. En oversættelse fra danske til tyske sætninger vil ikke være simpel på grund af verbernes forskellige placering i de to sprog.

Oversættere for Pascal er oftest konstrueret på denne måde, og denne type oversættere kaldes også *ét-passage-oversættere*. Strukturen af en typisk programoversætter er dog en anelse mere kompliceret end antydet ovenfor, da det er nødvendigt at overføre symboltabeller mellem deltræerne, jvf. kapitel 8.

13.3 Fejlbehandling

En anden ting, vi har gået let henover, er fejlbehandling. Datalogparseren beskrevet i afsnit 13.1 går ganske enkelt i stå, når den møder noget, der ikke hører hjemme i et korrekt Datalogprogram. Det er ikke særligt hensigtsmæssigt: Der er ingen indikation af, hvor fejlen befinder sig, og brugeren kan højst få oplysning om én fejl for hvert forsøg på at få sin tekst accepteret af parseren.

Det første problem — at lokalisere fejlen — kan løses ved at lade parseren producere en udskrift af teksten (på ældre edb-dansk, en »listning«), samtidig med den læser sig igennem den. I denne udskrift markeres så, hvilke symboler, der er malplacerede. Vi har tidligere set, at førstemængderne kunne hjælpe med til at karakterisere fejlen.

Det andet, at få parseren på benene efter en fejl, så den kan læse videre i teksten, er straks en mere delikat affære. Her findes ingen standardløsning, det er et spørgsmål om kvalificeret ingeniørarbejde og finjustering. Enhver, der har prøvet at arbejde med en oversætter for et sprog med en righoldig syntaks, Simula, f.eks., ved, hvad der tales om. Igen henvises til (Aho, Sethi, Ullman, 1986) for en mere fyldig behandling. Her antydes blot, hvordan man måske kunne have gjort for Datalog. Vi vil som et eksempel forestille os, at parseren er igang med at læse en klausul, og at den har læst et korrekt hovede og derefter møder et malplaceret symbol. Sådan som parseren er beskrevet i afsnit 13.1, vil den gå i stå med

```
error(":- eller . forventet").
```

Men når nu den er så genial, at den kan finde ud af, hvad der burde have stået, skulle man måske overveje at udnytte denne viden. Hvis nu det læste symbol lignede »`:-`«, f.eks. et enkeltstående »`:`«), er følgende måske en fornuftig reaktion: Markér fejlen i udskriften, og lad parseren tro, at alle tegn til og ikke med førstkommende bogstav rent faktisk var en »`:-`«. Hvis det symbol, der gav anledning til miseren, var et atom, så kan man måske antage, at brugeren har glemt at afslutte klausulen, som da skulle være et faktum, med et punktum. I såfald skulle parseren blot undlade at udføre et enkelt kald af `read_symbol` og fortsætte, som intet var hændt. Men på den anden side, hvis vor bruger blot har glemt at skrive »`:-`«, eller måske i sin enfold har skrevet »`if`« i stedet for? Osv. osv.

Opgave 13.1 I mange Prolog-systemer opfattes alt fra og med et »%«-tegn og hen til liniens afslutning som en kommentar. Hvordan vil du modificere klassen `LEXIKALSK_SCANNER`, så den tager højde for dette?

Opgave 13.2 Skitsér de leksikalske symboler, som optræder i Java. Hvilke af de forudsætninger, som lå bag udformningen af den leksikalske analyse for Datalog holder stadig — og hvilke holder ikke? Skitsér en algoritme til leksikalsk analyse for Java.

Opgave 13.3 Opskriv førstemængderne for alle syntaksdiagrammerne for Datalog.

Opgave 13.4 Udfør en håndsimulering af algoritmen til formatteret udskrift (side 209) på det Datalog-program, hvis kildetekst er som følger.

```
abe(ko,Kamel):- ged(Hest,gris,kylling), gås(and).
bas(ta).
```

Opgave 13.5 Skitsér, hvordan et program til formattering af Datalog-programmer kan konstrueres ved at erstatte parserens kald af `new` med kode fra `pretty_print`-procedurerne.

13.4 Bottom-up parsing

For at kunne forstå bottom-up-princippet, starter vi med at karakterisere top-down, så vi kan sammeligne de to. Idéen i top-down er, at parsealgoritmen foretager kvalificerede gæt på, hvilke syntaksregler, den givne symbolstreng måtte være genereret ud fra. Dette kvalificerede gæt foretages udfra det næste, ikke-læste symbol og i øvrigt udfra, hvilke konstruktioner, parseren allerede har genkendt, og dem, den »tror«, den er ved at genkende. Hvis en top-down parser for eksempel er i gang med at læse sætninger i et Simulaprogram, og det næste symbol er et **if**, så må den næste sætning nødvendigvis være en betinget sætning. Parseren anvender da reglen for betingede sætninger, og den kan tillade sig at forvente, at der kommer en betingelse efter **if**. Hvilken af syntaksreglerne for betingelser, som kommer i anvendelse, afhænger så af det første symbol i den faktisk forekommende betingelse.

Betegnelsen top-down kommer af, at et syntakstræ erkendes fra oven og nedefter. Vor antydede Simulaparser fra før, hvis den skal genkende et helt program, finder først ud af »dette har værsgo' at være et program«, derefter går den løs på de globale erklæringer, »her skal være en erklæring«, og måske »dette må være en procedureerklæring«. Hvis parseren var udstyret til at bygge et syntakstræ, så ville den først konstruere et programobjekt og derefter hægte de mere detaljerede undertræer på fra oven og ned.

Bottom-up virker den anden vej, i stedet for at bygge på postulater om, hvad de endnu ikke undersøgte symboler skal forestille, erkendes anvendelsen

af en syntaksregel først, når *hele* den tilhørende sekvens af leksikalske symboler har været undersøgt. Gentager vi eksemplet med Simula, vil en bottom-up-parser, starte med at konstatere »hmm, dette er et **begin**, det ligner ikke noget, det gemmer vi lidt«, derefter vil den gå videre i teksten og forhåbentligt genkende en erklæringsdel. Når den har gjort det, går den videre og genkender forhåbentlig hovedprogrammet som en sætning. Så må den læse videre, her står måske et **end**. »Hmm, nu har jeg set et **begin**, en erklæringsdel, et sætning og et **end**, hmm, det minder mig om et eller andet. Heureka, det er jo en blok!« Nå, derefter følger ikke mere tekst, dvs. et `END_OF_FILE`. »Et blok efterfulgt af en `END_OF_FILE`, hmm, det skulle vel aldrig være ..., jo, det er minsandten et program!!« Hvis en bottom-up-parser, udover at genkende teksten, også konstruerer et syntakstræ, så vil den naturligt starte med at bygge nogle små træer og, efterhånden som den har tilstrækkeligt af dem, gradvist sætte dem sammen til større træer.

Bottom-up-parserens virkemåde udviser en vis analogi med den måde, vi bygger korthuse på, hvorimod top-down mere ligner den måde træer og buske (måske snarere deres rodnet) vokser i naturen.

En mere praktisk sammenligning viser, at top-down giver anledning til de mest elegante parseprogrammer, som er nemme at skrive og vedligeholde i hånden. Til gengæld er metoden ikke særligt tolerant, hvad angår de grammatikker, den virker for. Betingelsen, at skulle erkende af hvilken art den efterfølgende symbol-sekvens er, udfra ét symbol, er meget restriktiv. Bottom-up-parsere konstrueres oftest som tabelstyrede algoritmer, hvor så al information om om den strukturelle syntaks ligger i en tabel, og algoritmen er den samme fra gang til gang. Disse tabeller konstrueres udfra en analyse af grammatikreglerne. Tabellerne har til formål, på en effektiv måde at afgøre, »skal dette næste symbol gemmes i interne strukturer til senere brug, eller er det afslutningen på et eller andet, vi kan reducere«. I eksemplet ovenfor, overvej forskellen i behandlingen af **begin** og **end**. (Det ville jo ikke være særligt effektivt, i hvert tilfælde, at løbe hele grammatikken igennem for at se, om der er noget, der matcher). Der gælder selvfølgelig også nogle begrænsninger på grammatikken, for at disse tabeller kan konstrueres, men de ikke er nær så restriktive som de tilsvarende for top-down. Ydermere har de kendte bottom-up-metoder den fordel, at de kan tilpasses tvetydige grammatikker, der som oftest er væsentligt enklere end ikke-tvetydige grammatikker. Tvetydighederne opløses ved at knytte en passende prioritering på de enkelte syntaksregler — hvilket vil blive illustreret nedenfor.

Parsergeneratoren YACC (Johnson, 1975, Aho, Sethi, Ullman, 1986, eller tilgængelig UNIX-dokumentation) fungerer vha. en bottom-up-metode.

Generatorens primære funktion er, ud fra en given grammatik, at konstruere tabeller for parseren. Hvis tabellerne ikke kan konstrueres, vil generatoren forsøge at informere brugeren om, hvordan grammatikken måske kan ændres, så den kan gå gennem nåleøjet. Såfremt det lykkes at konstruere tabellen, genererer YACC så yderligere et program, som indeholder dels den generelle bottom-up-algoritme og dels nogle semantiske aktioner, som angives i brugerens specifikation. YACC kan benyttes som en oversættergenerator. Til hver syntaksregel kan knyttes aktioner, som beskriver, hvordan den givne konstruktion skal oversættes — eller for den sags skyld fortolkes.

13.5 Et eksempel på en bottom-up-parser

Vi vil forklare princippet i bottom-up-parsing ud fra et eksempel. Nedenfor er givet en enkel og klar, omend tvetydig, grammatik for enkle regneudtryk. Vi vil underforstå den sædvanlige prioritering, dvs. at `1 + 2 * 3` betyder det samme som `1 + (2 * 3)`, og under ingen omstændigheder `(1 + 2) * 3`. Den er beskrevet vha. BNF-notation og ikke, som vi tidligere har brugt det, ved de noget behageligere syntaksdiagrammer. Det ligger der ikke noget principielt i, det gør det blot nemmere at forklare bottom-up parsing. Nonterminaler er skrevet i kursiv, alt andet er terminalsymboler/leksikalske symboler. De små numre er ikke en del af grammatikken, men vil blive brugt som reference nedenfor.

(1) *udtryk* ::= *udtryk* + *udtryk*
(2) *udtryk* ::= *udtryk* * *udtryk*
(3) *udtryk* ::= (*udtryk*)
(4) *udtryk* ::= *tal*

Vi beskriver først grundprincippet, anvendelse af de såkaldte reduktioner, og derefter skitseres det, hvordan dette kan implementeres vha. hensigtsmæssige datastrukturer.

13.5.1 Princippet: reduktion

Grundprincippet i bottom-up-parsing er hele tiden at erstatte sekvenser af terminal- og nonterminal-symboler, som matcher med højresider af syntaksregler, med den tilsvarende nonterminal fra venstre-siden. Denne operation kaldes en *reduktion*. Arbejdsgangen illustreres ved et eksempel. Vi starter med en sekvens af leksikalske symboler, som vi ved gradvise reduktioner redegør for, faktisk er et udtryk. Vi opfatter i første omgang alle symboler som hægtet

i en og samme liste, vi skelner ikke mellem, hvad der i en implementation kunne ligge på en fil og i interne datastrukturer. Vi betragter følgende tekststreng.

```
1 + 2 * 3 * 4 + 55
```

Vi antager, at en forudgående leksikalsk analyse har genkendt sekvenserne af cifre som tal, så strengen ud fra et syntaksgenkendelsessynspunkt altså ser således ud.

*tal + tal * tal * tal + tal*

I det følgende angives ved understregning hvilken delstreng, der bliver udset til reduktion, og hvilken regel, der reduceres med. At reduktionerne foretages i en rækkefølge, som netop afspejler den underforståede prioritering, kan for nuværende forstås som held eller magi, alt efter temperament.

*tal + tal * tal * tal + tal*	— reducer vha. regel 4:
*udtryk + tal * tal * tal + tal*	— reducer vha. regel 4:
*udtryk + udtryk * tal * tal + tal*	— reducer vha. regel 4:
*udtryk + udtryk * udtryk * tal + tal*	— reducer vha. regel 2:
*udtryk + udtryk * tal + tal*	— reducer vha. regel 4:
*udtryk + udtryk * udtryk + tal*	— reducer vha. regel 2:
udtryk + udtryk + tal	— reducer vha. regel 1:
udtryk + tal	— reducer vha. regel 4:
udtryk + udtryk	— reducer vha. regel 1:
udtryk	— slut!

I første linie udses første forekomst af *tal* til reduktion, vha. regel 4, til et *udtryk*. Dette *udtryk* indgår ikke i en delsekvens, som kan reduceres, så vi reducerer næste *tal* til et *udtryk*. Nu forekommer sekvensen *udtryk + udtryk*,

*udtryk + udtryk * tal * tal + tal*

og kan i princippet reduceres vha. regel 1, som antydet ved understregningen. Men det kan vi se er uklogt, da det ville stride mod den underforståede prioritering mellem operatorerne. At dette valg ikke er heldigt, kan mekanisk afgøres udfra det gangetegn, som følger efter det understregede udtryk — dette antyder lidt om parsetabellernes indretning, hvilket vi vender tilbage til senere.

Så i stedet gør vi altså noget andet. Vi må reducere nok et tal til et udtryk, hvorefter vi kan reducere vha. reglen for gange, regel 2. Og så fremdeles reducerer vi løs, indtil vi står tilbage med nonterminalen udtryk. Læser vi historien baglæns, ser vi en beskrivelse af, hvordan `1 + 2 * 3 * 4 + 55` kan genereres som et udtryk ud fra vor grammatik.

13.5.2 Effektiv implementation vha. en stak

Overfor beskrev vi princippet i bottom-up-parsing, nemlig successive reduktioner. Her vil vi angive en hensigtsmæssig datastruktur, som gør det muligt at udføre disse på rimeligt effektiv måde. Da programtekster ofte er meget lange og i forvejen ligger på filer, er det hensigtsmæssigt at lade de dele af teksten, som endnu ikke er undersøgt, blive liggende på filen så længe som muligt. Og i øvrigt organisere undersøgelsen, så den såvidt muligt foregår fra venstre mod højre. Hvad angår reduktionerne, så har de brug for en datastruktur, som er noget mere fleksibel end en fil. Vi kan her benytte vor gamle ven, stakken. Den tilhørende algoritme foretager så to ting,

- den kan flytte symboler fra ind-filen over på stakken, (kaldet en skifte-operation; dårligt oversat engelsk fra »shift«)

- den kan foretage reduktioner af de øverste elementer i stakken.

. På et givet tidspunkt skal algoritmen altså afgøre, om den skal udføre en skifte-operation eller en reduktion — og i givet fald hvilken.

I det følgende genfortælles analyse-historien ovenfor, blot med anvendelse af en stak, og hvor der nu også forekommer skifte-operationer i historien. Af hensyn til overskuelighed tegnes stakken med bunden til venstre. Bunden af stakken repræsenteres ved symbolet $\bullet$. For at få plads på siden forkortes nonterminalerne til et enkelt bogstav.

Stakken	Resterende ind-fil	Handling
•	*t* + *t* * *t* * *t* + *t*	skift:
• *t*	+ *t* * *t* * *t* + *t*	reducer vha. regel 4:
• *u*	+ *t* * *t* * *t* + *t*	skift:
• *u* +	*t* * *t* * *t* + *t*	skift:
• *u* + *t*	* *t* * *t* + *t*	reducer vha. regel 4:
• *u* + *u*	* *t* * *t* + *t*	skift:
• *u* + *u* *	*t* * *t* + *t*	skift:
• *u* + *u* * *t*	* *t* + *t*	reducer vha. regel 4:
• *u* + *u* * *u*	* *t* + *t*	reducer vha. regel 2:
• *u* + *u*	* *t* + *t*	skift:
• *u* + *u* *	*t* + *t*	skift:
• *u* + *u* * *t*	+ *t*	reducer vha. regel 4:
• *u* + *u* * *u*	+ *t*	reducer vha. regel 2:
• *u* + *u*	+ *t*	reducer vha. regel 1:
• *u*	+ *t*	skift:
• *u* +	*t*	skift:
• *u* + *t*		reducer vha. regel 4:
• *u* + *u*		reducer vha. regel 1:
• *u*		slut!

13.5.3 Lidt om parsetabeller

Hvorvidt en algoritme skal skifte eller reducere i en given situation afhænger af

- indholdet af stakken, og
- den resterende del af ind-filen.

I de fleste algoritmer nøjes man at se et enkelt symbol frem på ind-filen. Hvis f.eks. den øverste del af stakken indeholder *udtryk + udtryk*, er der to muligheder, enten er det

1. en forekomst af et plusudtryk, dvs. der skal reduceres, eller
2. det sidste udtryk er en del af et større gangeudtryk, som skal reduceres først, dvs. der skal aldeles ikke reduceres nu, så ergo skiftes.

Tilfælde 2 vil afsløres af, at næste symbol på ind-filen er et gangetegn. I alle andre tilfælde, kan vi uden fare reducere, dvs. tilfælde 1.

Dette lille eksempel viser princippet i anvendelse af tabellen til at styre parsealgoritmen. Det viser også noget om, hvordan tabellen konstrueres ud fra en analyse af

- grammatikreglerne, og
- ekstra-information om operatorernes indbyrdes prioritering.

Hvordan tabellerne faktisk konstrueres er et studium for viderekommende jvf. (Aho, Sethi, Ullman, 1986), men heldigvis er denne proces automatiserbar.

Af effektivitetshensyn undgår man yderligere hele tiden at lave mønstergenkendelse på den øverste del af stakken (f.eks. at checke om de tre øverste elementer danner mønstret *udtryk + udtryk*). I stedet benyttes en tilstandsmaskine, som holder rede på tilpas meget viden om, hvad der faktisk ligger på stakken. F.eks. kunne tilstand 7 netop svare til, at de tre øverste elementer er *udtryk + udtryk*. Så summa summarum styres algoritmen altså af to tabeller,

- en *handlingstabel*, som, givet et tilstandsnummer (dvs. en kortfattet rapport om stakkens tilstand) og et ind-symbol, returnerer besked om, hvad der skal ske, dvs. enten
  - skift, eller
  - reducer vha. regel nr. *n*

  (eller eventuelt, at der er tale om en fejl i inddata).
- en *tilstandsstabel*, som givet et tilstandsnummer og et ind-symbol, returnerer et nyt tilstandsnummer.

Men som sagt, denne tilstandsmaskine er udelukkende en effektivisering, som ikke er nødvendig for at forstå princippet, så dette tema lader vi ligge.

For de, som måtte være interesseret, gives her en parsetabel (svarende til den omtale handlingstabel) for vor lille grammatik; den tager højde for såvel parantes-udtryk som den naturlige prioritering mellem plus og gange. Det vil ikke blive begrundet yderligere, hvorfor tabellen ser ud som den gør, og interesserede opfordres til at foretage håndsimulering af parsing på udvalgte eksempler. Der er angivet handlinger svarende til givne mønstre for den øvre del af stakken og tilhørende ind-symboler. Hvor der blot står en streg for ind-symboler, betyder det, at handlingen skal udføres uafhængigt af, hvilket ind-symbol, der er tale om. Symbolet *eof* er en forkortelse for `END_OF_FILE`, dvs. ind-filen er ikke længere. Når algoritmen frem til en kombination af stak-indhold og ind-symbol, hvor der ikke er en indgang i tabellen, er der tale om

en syntaksfejl. Tabellens første indgang refererer til den tomme stak markeret ved •.

Øverste stak-elem.	Ind-symbol(er)	Handling
•	*tal* (	skift
• *udtryk*	*eof*	slut
• *udtryk*	—	skift
· · · +	—	skift
· · · *	—	skift
· · · *udtryk* + *udtryk*	*	skift
· · · *udtryk* + *udtryk*	*eof* +)	reducer vha. regel 1
· · · *udtryk* * *udtryk*	—	reducer vha. regel 2
· · · *tal*	—	reducer vha. regel 4
· · · (	—	skift
· · · (*udtryk*)	—	reducer vha. regel 3

13.6 Bottom-up parsing og oversættelse

Vi forklarer her, hvordan en bottom-up-parser kan udvides til at fungere som en oversætter svarende til de oversættelsesregler formuleret vha. Prolog, som vi beskriv i afsnit 4.4.3; metoderne i nærværende afsnit kan benyttes, når der er brugt for mere effektive implementationer.

Vi benytter her en generel notation til at specificere syntaksstyrede oversættelser, som også kan bruges, f.eks. i forbindelse med top-down. Vi minder om, at en syntaksstyret oversættelse er en oversættelse fra et sprog til et andet. Dvs. til hver frase i »kildesproget« svare præcis en frase i »målsproget«, og oversættelsen af en given frase er bestemt ved en sammensætning af oversættelserne af dens delfraser — og reglen for denne sammensætning afhænger ene og alene af den anvendte syntaksregel. Altså, til hver syntaksregel hører netop en oversættelsesregel. I en oversættelse fra aritmetiske udtryk til kode for en stakmaskine, for eksempel, kan vi beskrive, hvordan et plusudtryk skal oversættes, på følgende måde.

> »Oversættelsen af et plus-udtryk består oversættelsen af det første deludtryk hægtet sammen med oversættelsen af det andet udtryk efterfulgt af maskin- instruktionen ADDÉR.«

Vi kan med fordel bruge en mere formel notation illustreret som følger.

> *udtryk* ::= $udtryk_1$ + $udtryk_2$
> → $udtryk_1$ $udtryk_2$ ADDÉR

Efter syntaksreglen, adskilt af pilen, følger den tilsvarende oversættelsesregel. Vi har nummereret nonterminalerne på højresiden for at kunne kende forskel på de to med samme navn. Så hvis det første del-udtryk oversættes til

```
STAK 7
```

og det andet til

```
STAK 2
STAK 5
GANG
```

så oversættes det sammensatte udtryk til

```
STAK 7
STAK 2
STAK 5
GANG
ADDÉR.
```

Skal vi være helt formelle, så refererer forekomsten af nonterminalerne til højre for pilen altså til oversættelserne af de respektive delfraser, og der forekommer implicitte tekstsammensætningsoperationer mellem de enkelte elementer (i eksemplet ovenfor, mellem *udtryk*$_1$ og *udtryk*$_2$ og mellem *udtryk*$_2$ og ADDÉR).

Med den antydede notation kan vi beskrive enhver syntaksstyret oversættelse, bl.a. simple oversættelser som vi har set på tidligere i forbindelse med top-down, hvor nonterminalerne i oversættelsesreglen forekommer i samme rækkefølge som i syntaksreglen, og andre oversættelser, hvor man bytter om på rækkefølgen. Det sidste ville nok være relevant i en oversættelse fra tysk til dansk.

Her følger vor grammatik udvidet med oversættelsesregler, som oversætter regneudtryk til kode for en abstrakt stakmaskine.

(1) *udtryk* ::= *udtryk*$_1$ + *udtryk*$_2$
→ *udtryk*$_1$ *udtryk*$_2$ ADDÉR

(2) *udtryk* ::= *udtryk*$_1$ * *udtryk*$_2$
→ *udtryk*$_1$ *udtryk*$_2$ ADDÉR

(3) *udtryk* ::= (*udtryk*)
→ *udtryk*

(4) *udtryk* ::= *tal*
→ STAK *tal*

Vi har tidligere set, at den type oversættelser, som kaldes simpel, er velegnet i forbindelse med top-down parsing. De rekursivt nedstigende parseprocedurer kunne udskrive oversættelsen direkte uden at opbygge syntakstræer eller andre snedige datastrukturer. At en oversættelse er simpel, betyder, at nonterminalerne i oversættelsesreglen optræder i samme rækkefølge som i syntaksreglen.

Vi vil overveje en tilsvarende klasse af oversættelser for bottom-up-parsing, dvs. hvor oversættelsen kan skrives direkte ud undervejs. Altså, en bottom-up parser fungerer på den måde, at for en given konstruktion genkendes (og oversættes) delkonstruktionerne først, derefter erkendes arten af den hele konstruktion. Og først da kan der udskrives den del af oversættelsen, som er specifik for netop den anvendte syntaksregel. Så regler af form som (1), (2) og (3) i den ovenfor specificerede oversættelse overholder altså disse begrænsninger. Regel (4), derimod, er ikke så god, skulle vi holde os strengt til teorien, måtte vi så tilpasse vor stak-maskine, så man kunne skrive »5 STAK« i stedet for »STAK 5« som oversættelse at et udtryk, som bestod af tallet »5«.

13.7 Præcedensparsing

Et særligt specialtilfælde af bottom-up-parsing benyttes i forhold til de såkaldte præcedensgrammatikker. En præcedensgrammatik består af rækker af definitioner af operatorer som kan være præfiks, infiks og postfiks, ganske som vi kender det fra Prolog, og vi vil betragte en samling operatordefinitioner i Prolog som en prototypisk præcedensgrammatik.

En parser for Prolog kan opbygges som et bottom-up-parser, hvor afgørelsen af, hvorvidt der skal skiftes eller reduceres, foretages ved at kigge på de indgående operatorers type (f.eks. `xfy` o.lign.) og deres præcedenstal. — Det er en forholdsvis overkommelig opgave at konstruere en parser efter disse principper; dette overlades som en øvelse til den interesserede læser.

Fordelen ved en tabelstyret implementation (f.eks. Prolog-fakta om de aktuelt definerede operatorer) er, at tabellen og dermed parseren kan udvides »on the flight«, som vi kender det fra Prolog.