Henning Christiansen

Sprog og abstrakte maskiner

3. reviderede udgave

ROSKILDE UNIVERSITETSCENTER, DATALOGI — MODUL2, 2000
UÆNDRET GENOPTRYK 2012

Sprog og abstrakte maskiner

3. reviderede udgave

Henning Christiansen

3. reviderede udgave
januar 2000; uændret genoptryk © 2012
E-post: henning@ruc.dk

Indhold

1 Indledende om datalogiens mission og væsen

»... the art of programming is the art of mastering complexity, of mastering multitude and avoiding the bastard chaos as effectively as possible«

E. W. Dijkstra, 1972

Datalogiens genstandsområde er befolket med komplekse systemer, og en af udfordringerne i datalogien er at kunne håndtere komplekse systemer. Det er datalogien selvfølgelig ikke alene om, det foregår i så mange andre videnskaber. Julius Cæsar formulerede princippet »del og hersk« i forhold til det at holde sammen på et imperium, blot for at nævne et iøjnefaldende eksempel. Det, som gør datalogien til noget specielt, er den teknologi-historiske hændelse, at datamaskinen er opfundet. Lad os da forsøgsvis analysere, hvad nyt eksistensen af et sådant apparat kaster på banen.

Datamaskiner — eller computere, som de kaldes på nudansk — er indretninger, som arbejder med beskrivelser. Et program er en eksempel på en beskrivelse, f.eks. en beskrivelse af en række beregninger, som ønskes udført. Der er også mange andre videnskaber, som beskæftiger sig med beskrivelser som en del af deres genstandsområde, f.eks. litteraturhistorie, lingvistik og matematik. — Datalogiens beskrivelser har nogle helt specielle egenskaber.

1. Der er tale om *formelle* beskrivelser, formel forstået derhen, at der er en entydig sammenhæng mellem udtryk og indhold. Det er ret ekstremt. Juridiske formuleringer og matematikbøger er i sammenligning fyldt med upræcision, skjulte forudsætninger og muligheder for forskellige fortolkninger, og skulle der være et par små fejl eller fem, fungerer helheden alligevel. Ændres omvendt et enkelt komma i program, kan dette meget vel ændre programmets betydning radikalt.[1]

[1] Var der ikke noget med en rumfærge, som de ikke kunne få til at lette, fordi der manglede et mellemrum i et Fortran-program?

2. Der er ofte tale om totalt set ekstremt komplicerede beskrivelser. Hvor mange programlinier og design af digitale kredse ligger der ikke til grund for, at jeg kan sidde og skrive denne tekst på min Mac, se den omgående reflekteret på min skærm, få den stavechecket og skrevet nydeligt ud på en laserprinter?

3. Beskrivelserne udsættes for mekanisk behandling, vi har maskiner som kan »gøre«, hvad der står i beskrivelserne. Et program eller input til et program er en specifikation af en helt bestemt form for beregning, hvor maskinen udfører denne beregning.

4. Datalogiens emner er fyldt med refleksion og tilsyneladende cirkularitet. Et program er noget, som beskriver en sammenhæng mellem noget input og noget output. Disse objekter er igen blot beskrivelser, så i en passende forstand er et program en beskrivelse af, hvordan beskrivelser afbildes over i beskrivelser.[2]

Man kan sige, at 1 og 3 tilsammen gør matematik til virkelighed. Datamaskiner er apparater, som (når de ellers fungerer) opfører sig efter kendte matematiske regler. Symbolske beskrivelser (programmer) omsættes efter veldefinerede principper til processer. Sammen med 4 kan man så overveje, hvad man egentlig kan udtrykke i et program, hvilke former for beregninger man dybest set kan foretage på en datamaskine — og hvilke ikke. Problemstillinger som denne leder tilbage til arbejde gjort indenfor matematisk logik, noget af det mest i iøjnefaldende arbejde er gjort af Gödel, Turing og Church.

Ser vi på pkt. 2, sammenholdt med de andre punkter, så må vi have en eller anden måde at nedbryde kompleksiteten på, og som i alle andre af livets forhold, må vi sætte vor evne til abstraktion ind. (Og det er vel en »evne« datamaskiner ikke er i besiddelse af!)

Definition. At *abstrahere* er at fokusere på de egenskaber af virkeligheden, vi i den givne sammenhæng er interesseret i.

Opgave 1.1 Hvordan vil du forklare ordet »abstrakt« i betegnelsen »abstrakt kunst«?

Derfor tager vi i denne bog udgangspunkt i begrebet *abstrakt maskine.* Den Mac, som står foran mig, er svær at ræsonnere om, hvis jeg tager det hele med. Lige nu opfatter jeg den som en tekstbehandlingsmaskine, som jeg »programmerer« med tekstfragmenter og diverse kommandoer, flytninger af en pil ved hjælp af en mus osv. Hvis jeg, mens jeg sad og skrev denne tekst, skulle tænke på, at der lå et operativsystem nedenunder, som på den og den måde modtager mine tegn og sender dem gennem diverse buffere hen til det

[2] Blot for at nævne et eksempel, så er de fleste Pascal-oversættere skrevet i Pascal.

program osv. osv., så ville jeg hurtigt miste overblikket.[3]

I går skrev jeg et Prologprogram på min Mac, og på det tidspunkt tænkte jeg på den som en Prologmaskine. Og hvilke opfattelse har de programmører, som har lavet Prologsystemet eller operativsystemet, eller det, der får grafikken og musen til at fungere, haft — hvordan har de opfattet maskinen, og hvilke nye maskinopfattelser har de ønsket at skabe illusion om?

Formålet med denne bog er at give en sammenfattende forståelse af alle disse lag, sprog og systemer som dukker op i forbindelse med det fænomen, vi almindeligvis kalder »den moderne computer« — en teoretisk forståelse til et niveau, som man kan forvente indgår i bachelor og kandidatuddannelser i datalogi og andre IT-orienterede retninger. I en hovedfagsuddannelse i datalogi, eller for studerende i øvrigt med en interesse for det teoretiske eller (f.eks. oversætter-) tekniske, kan det være nødvendigt at gå videre med anden litteratur.

I bogen er lagt tilrette med henblik på at henvende sig til en bred kreds af studerende, både med hensyn til deres interesser og deres baggrund. For studerende med en karriere i tankerne indenfor systemudvikling, hvad enten det handler om udvikling af komplekse systemer eller mere bløde, designprægede aktiviteter, skulle bogen gerne give dem så meget overblik, at de kan udvikle eller designe bedre systemer — fordi de, om man så må sige, får en bedre forståelse af de grundlæggende egenskaber ved det materiale, de arbejder med. Der er derfor også lagt vægt på at fremhæve, at bogens abstrakte maskiner og definerende fortolkere er praktisk værktøj til design, prototypeudvikling og eksperimenteren med grænseflade- og andre sprog.

Bogen kan læses af studerende med forskelligartede baggrunde og er tænkt at kunne appellere til personer med eksempelvis en humanistisk baggrund, ligesåvel som den skulle gå ind hos dem med en naturvidenskabelig baggrund. Principperne, som er anvendt i bogen (og beskrevet nedenfor) er udviklet over en årrække i forbindelse med undervisning på overbygningsuddannelsen i Datalogi på Roskilde Universitetscenter, hvor der netop optages studerende fra alle hovedområder, naturvidenskab, humaniora og samfundsfag. Denne bog bygger på en påstand om, at man altså godt kan ramme denne brede målgruppe med et fælles sprog, og holde alle studerende engagerede — når blot det gøres på den rette måde, og denne bog er et bud, hvordan dette kan gøres.

Den metode, jeg har anvendt, er at give et blødt teoriapparat baseret på abstrakte maskiner og kombinere det med konkrete eksempler, hvor jeg be-

[3] Analogi: Hvilken skæbne ville vi ikke forudse for den cyklist, som baserede sit venstresving på overvejelser om Newtons fysik — i stedet for bare at dreje om hjørnet?

nytter logikprogrammeringssproget Prolog som beskrivelsessprog. Fordelene ved Prolog i denne sammenhæng er, at sprogbeskrivelserne samtidigt også er kørende prototyper for de beskrevne sprog. De studerende kan så afteste eksempler, og hvis de ikke er tilfredse med de eksemplificerede sprog, så kan de jo bare lave om på dem og se, hvad resultatet så bliver. Envidere har Prolog den oplagte fordel fremfor diverse matematiske vokabularier, at den studerende kan basere sin forståelse på den præcise intuition, som opnås gennem praktisk erfaring med et programmeringssprog. I denne forbindelse er det værd at hæfte sig ved, at Prolog derudover er et af de få programmeringssprog som samtidigt har en intuitivt og matematiske simpel, deklarativ semantik — der er, om man så må sige, matematik forklædt som et tilgængeligt og appetitligt programmeringssprog. Og her udnyttes naturligvis den appeal Prolog derudover har i forhold til en eksperimenterende og interaktiv programmeringsstil.

Selvom jeg omtaler begrebsapparatet som »blødt«, er det tænkt at være stringent i den forstand, at det for den matematisk indstillede læser vil være nemt at fylde de formelle detaljer ind, og der lægges heller ikke skjul på, at det grundlæggende bygger på logik, mængde- og funktionslære (og disses moderne revisioner i form af gitter- og domæneteori). Opblødningen ligger til gengæld i præsentationen, hvor der lægges vægt på de begrebsmæssige strukturer mere end den matematiske præcision (som den matematiske læser som sagt selv kan tage vare på).

Den abstrakte maskine fungerer som et samlende begreb, som måske kan være med til at give en mere sammenhængende forståelse af de mange, tilsyneladende vidt forskellige »systemer«, som man støder på. Jeg lægger stor vægt på at fremhæve, at det egentligt ikke skulle være ny tankegang for læseren, — det er blot principperne i struktureret programmering anvendt i en større sammenhæng! Derudover benyttes anledningen til at fremvise nogle teoretisk interessante, hypotetiske maskiner, som i tidens løb er anvendt i mere filosofiske overvejelser om, hvad datamaskiner kan og ikke kan. Bogen kan passende læses sammen med eller som introduktion til (Tanenbaum, 1999). Tanenbaums bog om den tekniske opbygning af en computer med samt dens operativsystem m.v. er ubetinget den mest velskrevne lærebog på sit område, men til gengæld kan den måske forekomme en smule teoriløs. Nærværende bog supplerer yderligere Tanenbaum ved også at kigge på de højere lag, f.eks. programmeringssprog, databasesprog og brugergrænsefladesprog, hvor Tanenbaum standser ved det symbolske maskinsprog. Hvor og hvornår begrebet af abstrakte maskiner første gang er blevet præciseret er ikke klart for mig — betegnelse optræder i hvert fald i den særdeles læseværdige artikel, det indledende citat er hentet fra.

Denne bog kan også bruges som appetitvækker til den spændende bog (Abelson, Sussman, 1985), som på mange måder har en indfaldsvinkel som ligner min, men som er langt mere teoretisk orienteret og på denne front er langt mere grundig — og adskiller sig ved at benytte funktionssprogrammering lidt i samme ånd, som jeg her benytter logikprogrammering.

Slutteligen skal det pointeres, at bogen er bedst egnet til brug i en sammenhæng, hvor der er plads til diskussion med medstuderende og en personlig formidling fra forelæsere og hjælpelærere. Bogen giver nemlig ikke altid standardsvar eller fuldstændige definitioner, men trækker ofte på »generel erfaring« og lægger op til diskussion. Opgaverne i bogen er en blanding af praktiske opgaver af varierende omfang og diskussionsopgaver, som understreger et aspekt i teksten. Det er en god idé at reflektere over opgaverne under læsningen, også selv om man ikke løser dem her og nu. For at få den fulde forståelse er det væsentligt, at man arbejder praktisk med opgaverne, specielt med de små oversættere, fortolkere og andre sprogbeskrivelser formuleret i Prolog — det er først derigennem man for alvor får internaliseret begreberne.

En kort oversigt

I kapitel 2 defineres, hvad der forstås ved en abstrakt maskine og der gives et bredt udvalg af eksempler, som dækker spillerummet fra struktureret opbygning af programmer til modeller, som afspejler den fysiske maskines egenskaber. Kapitel 3 beskriver på overordnet måde begrebet implementation, dvs., hvordan en abstrakt maskines instruktioner på forskellig vis kan »afbildes« over i instruktioner for en anden abstrakt maskine, som måske ligger lidt tættere på den »fysiske« maskine. Implementation er her et middel til at binde de forskellige lag sammen — specielt fremhæves oversættelse, fortolkning og abstraktionsmekanismer i programmeringssprog. Disse begreber er også centrale i Tanenbaums bog.

Kapitel 4 anlægger en sproglig synsvinkel og beskriver, hvad der forstås ved syntaks, semantik og pragmatik. Der lægges også vægt på at gøre begrebet metasprog eksplicit: Metasproget er det, vi benytter, når vi beskriver et eller andet andet sprog, vi har fået interesse for. Vi præsenterer her traditionelle beskrivelsesværktøj for syntaks på den ene side, og på den anden viser, hvordan Prolog benyttes som metasprog for semantik. Vi udsætter i denne forbindelse Prologs indbyggede faciliteter til meta- og refleksiv programmering for en nærmere granskning. Endelig viser vi Prologs indbyggede grammatiknotation, definite klausulgrammatikker, som trods visse ulemper er et populært metasprog i forhold til specielt analyse af forenklet naturligt sprog, men

som også kan bruges til programmerings- og andre »computersprog«.

Derefter ser vi, i kapitlerne 5–9, på forskellige arter af sprog. Først logiske kredsløb, som kan afbildes elegant over til Prologprogrammer, og dette eksempel tjener ligeså meget til at vise Prolog in action. Dernæst tager vi skridtet til maskinsprog (hvor vi lægger vægt på den sekventielle kontrol) og til while-programmer. Vi benytter en delmængde af Lisp til at eksemplificere funktionsprogrammering og benytter anledningen til at dvæle lidt ved detaljer omkring parameteroverførsel. I kapitel 8 ser vi på typer og erklæringer af procedurer og klasser, og analysere de deraf affødte kontekstafhængige, syntaktiske begrænsninger. Endelig ser vi i kapitel 9 på relationel algebra og sammenholder det med den delmængde af Prolog, der kaldes Datalog; semantikken af relationelle udtryk beskrives gennem en evaluator.

I kapitel 10 tager vi en teoretisk drejning og ser på grundlæggende egenskaber som beregnelighed og universalitet. Her refererer vi (naturligvis) til Turing-maskinen, som repræsenterer en prototypisk datamaskine brugt i studiet af, hvad man kan beregne og hvad man ikke kan beregne på en datamaskine. Turing-maskinen, som stammer fra 1930'erne, hører til blandt de første skridt på vej mod en videnskab, som senere blev kaldt datalogi. I denne videnskabshistoriske sammenhæng kunne det også være relevant med en omtale af von Neuman-maskinen, men den er udmærket skitseret hos f.eks. (Tanenbaum, 1999, kap. 1). — von Neuman-maskinen er den læst, over hvilken alle udbredte, moderne datamaskiner er bygget.

Kapitel 11 svinger over i den pragmatiske afdeling ved at studere værktøjer, som en programmør kan have nytte af udover oversættere og fortolkere. Som eksempler fremhæver vi tracere, debuggere og værktøj til effektivitetsmåling. Sådanne kan overraskende nemt karakteriseres ved at tilpasse de allerede beskrevne fortolkere eller ved banale »source-to-source« oversættelse.

I kapitel 12–14 skifter vi stil og studerer effektive og praktisk relevante metoder til syntaksgenkendelse — leksikalsk analyse og parsing — og viser gennem udvikling af et fungerende Datalog-system i et objektorienteret sprog, hvordan disse metoder kobles sammen med en fortolker. I appendiks vises den samlede kildetekst hertil i to versioner, i det aktuelle Java og det historisk vigtige Simula.

Hvad der kunne komme med i næste udgave

Denne bog benyttes aktuelt på Roskilde Universitetscenters overbygningsuddannelse i datalogi, på andet moduls kursus »Sprog og logik«, sammen med Bratkos Prolog-bog som de to centrale værker. Der bygges på den forudsæt-

ning, at de studerende er fortrolige med objektorienteret programmering, aktuelt i Java. Emnevalget i bogen afspejler også, at de fleste studerende samtidigt følger kurser om databaser og evt. maskinarkitektur og netværk.

Bogen er tænkt som et forstudium til en egentlig lærebog, og man kunne her foreslå den udvidet med en række emner: En selvindeholdt introduktion til Prolog (som ikke udelukker sideløbende brug af en lærebog), en kort oversigt over mængdelære og logik, indenfor det programmeringssproglige område vil det være på sin plads at give en mere fyldestgørende beskrivelse af funktionsprogrammering og nyere tiltag indenfor logikprogrammering (bl.a. constraint-logik). En indplacering af objektorienterede sprog i vor begrebsramme vil bestemt heller ikke være af vejen, og for at understrege bredden af, hvad vi forstår ved sprog, kunne man kigge på teksformatteringssystemets TeX's inputsprog, som indeholder mange interessante forcer og særheder. Nye afsnit, som illustrerer principper for implementation af rekursive procedurer i traditionelle programmeringssprog er under overvejelse ligesom en beskrivelse af, hvordan logikprogrammeringssprog implementeres i praksis. Et kapitel, som helliger sig den historiske udvikling, er også på sin plads — fra Aristoteles, Euklid og Pythagoras, over den arabiske blomstring, renæssancens idealer, 1800-tallets kombination af romantik, industrialisme og formel logik, pausende ved en af Hilberts teser anno 1900, via Gödel, Turing, Church, og frem til den seneste datalogiske forskning. I den endelige udgave vil hver kapitel blive forsynet med en koncis opsummering, og der vil blive lagt større vægt på at give fyldestgørende, bibliografiske oplysninger. Endelig må det tilstås — og jeg håber læseren kan bære over med det — at bogen trænger til en begrebsmæssig afpudsning, udlugning af gentagelser og en sproglig efterbehandling.

Tak for hjælp og inspiration

Tak til Jens Frandsen, som har udviklet TeX-formatet anvendt i bogen, og har konverteret diverse dokumenter til TeX og har produceret de nydelige figurer ud fra mine håndtegnede skitser. Til Troels Andreasen for et inspirerende samarbejde og udvikling af modul 2-kurset i Datalogi på Roskilde Universitetscenter; kapitel 8 om relationel algebra er baseret på en større opgave, de studerende har skullet løse, hvor Troels har bidraget med introduktionen til datamodeller og relationel algebra (hvilket faktisk henledte nærværende forfatter på, at databaser også er interessante!). Tak også til Keld Helsgaun, som har omskrevet min implementation af Datalog-sproget fra Simula til Java. Og naturligvis til de talløse skarer af studerende som har måtte lide under tidlige-

re versioner af notater og småbøger, som er samlet i denne bog — og som har bidraget med megen inspiration, som diskussionspartnere og forsøgskaniner.

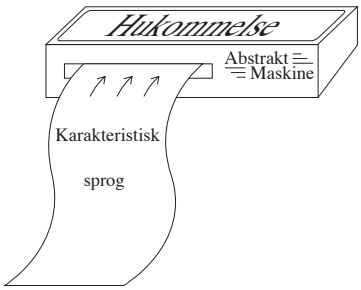
2 Abstrakte maskiner, definition og eksempler

Vi giver først en passende, ikke alt for præcis definition af, hvad en abstrakt maskine er. Derefter følger et antal eksempler, som vel egentlig skulle være læseren bekendt allerede. Det er blot det, at betragte dem som instanser af det samme begreb, som kan være nyt.

2.1 Vort grundbegreb

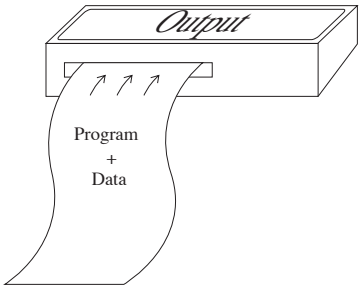
Definition. En *abstrakt maskine* er karakteriseret ved

- et *karakteristisk sprog*, som er en samling af sentenser eller fraser,
- en *hukommelse*, som på et givet tidspunkt rummer en værdi fra en eller anden mængde af mulige værdier,
- en *semantisk funktion*, som er en afbildning fra sentenser i det karakteristisk sprog og (gammel) hukommelse til ny hukommelse,
- evt. andre egenskaber.



Det karakteristiske sprog svarer til de legale input til maskinen. Vi har ikke beskrevet noget specielt vdr. muligt output — skulle der være brug for dette,

så opfatter vi det som en del af hukommelsen. Man kan så at sige kigge ind i hukommelsen. Hvis det karakteristiske sprog indeholder faciliteter, som har karakter af indlæsning, kan vi antage, at det karakteristiske sprog består af korrekte programmer med efterhængt inputfil. Når vi har brug for det, vil vi notere den slags sammensætninger med et +, f.eks. *Mit-program* + *mine-data*.



Da der er tale om abstrakte — ikke nødvendigvis fysiske — maskiner, udtrykker den semantiske funktion nogen om intensionen med maskinen, ogikke så meget om, hvordan den faktisk er konstrueret. Der ligger også i betegnelsen abstrakt maskine, at den ikke nødvendigvis »eksisterer« (hvad det så end vil sige). En abstrakt maskine kan være en forenklet fremstilling af et faktisk, fysisk fænomen — eller den kan være af rent hypotetisk natur, som et middel til at ræsonnere om eller formidle datalogiske begreber. I eksemplerne i det følgende vil vi se abstrakte maskiner, som befinder sig forskellige steder i dette spektrum.

Læg også mærke til, at der hersker en dualitet mellem begreberne abstrakt maskine og sprog. Forskellen i betragtningsmåde er, at når vi fokuserer på sproget, ser vi på en mængde af sentenser, hver med en bestemt, universel betydning — når vi taler om abstrakte maskiner, tænker vi mere på noget, der foregår over tid, at maskinen indlæser en sentens, udfører en proces (bestemt ved den semantiske funktion), som repræsentere dens universelle betydning i en eller anden forstand. Det, vi kaldte »andre egenskaber« kan f.eks. handle om, hvor lang tid, den semantiske funktion måtte være om at udføre programmer — hvis det altså er det, man er interesseret i.

Det er også vigtigt at notere, at definitionen ikke udtaler sig om, hvordan sprog, hukommelse, værdier og semantisk funktion er specificeret — eller om de overhovedet er specificerede. Og da slet ikke noget om, at de skal være implementerede — eller realiserbare i nogen filosofisk eller teknisk forstand.

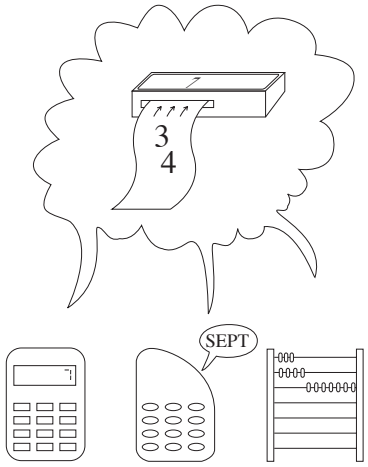
Når vi har brug for det, vil vi benytte en fast opstilling til at beskrive en abstrakt maskine. Her følger et lille eksempel.

Maskine Summations-apparat
Input-sprog: sekvenser af tal og instruktionen RESET
Hukommelse: en celle, som holder en talværdi
Semantisk funktion: Tal forstås som instruktioner, og instruktionerne i en input-sentens udføres i rækkefølge. RESET nulstiller hukommelsen, et tal adderes til hukommelsen.
eniksaM

Eksempler på udførelser givet af maskinen Summations-apparat:

Input:	Hukommelse-før:	Hukommelse-efter:
1	7	8
RESET 2 3 4	117	9
1 2 3 4 RESET	15	0

Dette er en meget simpel maskine, men den er dog relevant. Den giver en abstrakt (videnskabelig?) måde at betragte, f.eks. en (del af en) lommeregner, eller et program, nogen har skrevet, som kører på en helt bestemt datamaskine, som læser tal fra tal-tastaturet og præsenterer resultatet i form af lysende prikker på skærmen eller måske ved at sende passende koder til et talegenereringsmodul. Eller det kunne være et pædagogisk eksempel i en populær fremstilling om forskellen på matematik og datalogi. Men hvorom alting er, den abstrakte maskine eksisterer som begreb. i kan ræsonnere om summationsapparater uden at involvere nogen sen-industrielle frembringelser.



2.2 Programmeringssprog som abstrakte maskiner

Vor definition dækker også højere programmeringssprog:

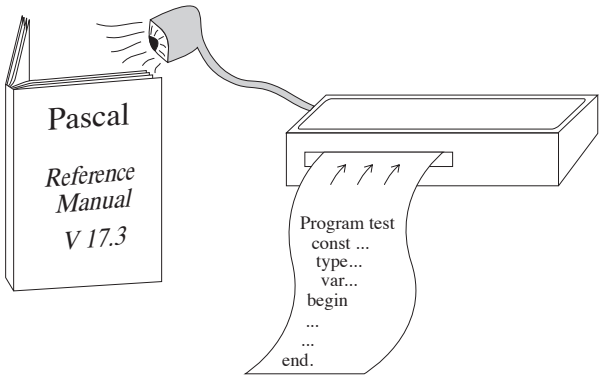
Maskine Pascal-maskine

Input-sprog: Et legalt Pascalprogram, jvf. grammatikken i en referencemanual, samt en input-fil for dette program.

Hukommelse: En output-fil.

Semantisk funktion: Sætter hukommelsen til at indeholde det output, som programmet ifølge referencemanualen skal producere.

eniksaM



Denne abstrakte maskine kunne måske modellere præcis samme fysiske fænomen, som det foregående Summations-apparat. Hvis nu min Mac havde Pascal liggende, kunne jeg skrive et program, som indlæste og udførte »summationsprogrammer«. Alt afhængigt af, hvad jeg måtte være interesseret i, kan jeg så ræsonnere om min Mac (dvs., det fysiske fænomen, som står foran mig) som Pascalmaskine, summationsmaskine (eller...). Det er egentligt også ret smart, at jeg kan forudsige, hvad der kommer ud af at køre bestemte programmer i summationssproget på min maskine uden overhovedet at vide noget om, hvordan det omtalte Pascalprogram måtte være skrevet, hvordan Pascaloversætteren er konstrueret, ned til, hvilke elektriske strømme, som faktisk løber rundt inde i kassen.

Opgave 2.1 Hvad betyder ordet »faktisk« i linien ovenover?

Hvis nu jeg er bruger og tror på, summationen fungerer rigtigt, så behøver jeg ikke at vide mere. Hvis jeg er programmør og skal lave en summationsmaskine, og jeg tror på, at Pascal fungerer nogenlunde korrekt på maskinen, så tænker jeg på den som en Pascalmaskine og behøver ikke at vide mere. Er jeg oversætterkonstruktør og skal lave en Pascaloversætter ... osv.

2.3 Struktureret programmering

De foregående eksempler beskrev maskiner, som på en eller anden måde eksisterede, før jeg begyndte at beskrive dem. Nogle af de mest nyttige, abstrakte maskiner er måske dem, som ikke svarer til noget, som eksisterer fysisk endnu. Trinvis forfining vha. pseudoprogrammer (Wirth, 1971), som vel alle programmører i en eller anden forstand betjener sig af, er et eksempel på kvalificeret brug af abstrakte maskiner.

Betragt følgende pseudoprogram.

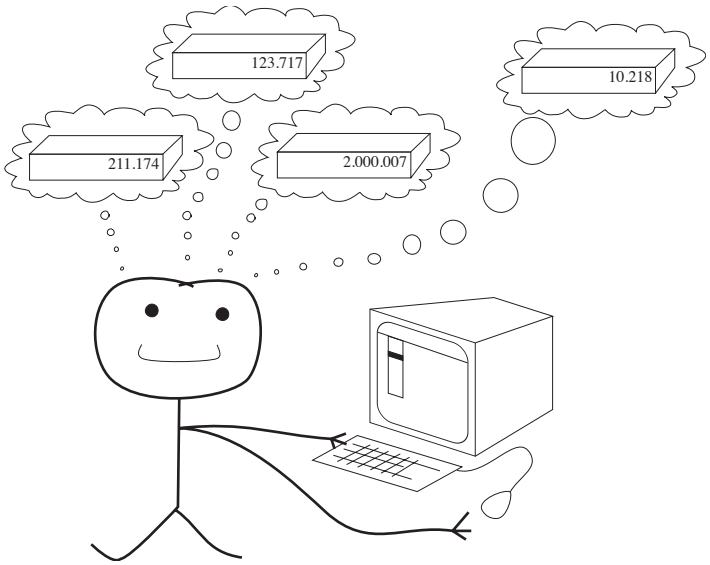
```
begin
  start;
  midt;
  slut
end
```

I den givne programmeringsproces har man en vis, måske endnu vag forståelse af, hvad de abstrakte instruktioner egentlig står for, men intuitivt set, så er der tale om et program. Programmet her er tydeligvis ikke skrevet i et »eksisterende« programmeringssprog og kan ikke udføres af nogen »eksisterende« maskine (endnu da).

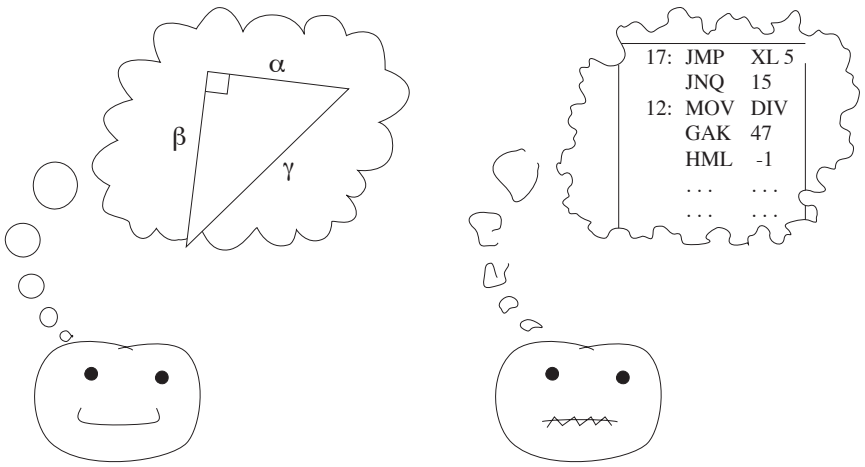
Maskine Programmør Jensens pseudomaskine nr. 10.217.
Input-sprog: Sekvenser af instruktioner start, midt og slut.
Hukommelse: Et lager sådan og sådan . . .
Semantisk funktion: Instruktionerne udføres i rækkefølge, start gør . . . ved hukommelsen, midt gør . . . og slut gør . . .
eniksaM

I den videre nedbrydningsproces handler det om, at udtrykke betydningen af pseudokodens instruktioner i andre pseudoinstruktioner, som er »knapt så abstrakte«, disse igen i andre instruktioner, så vi til sidst når ned til instruktioner i en maskine, som vi kan påstå »eksisterer«. Så næste skridt i programmeringsprocessen bliver måske den mentale konstruktion af følgende maskine.

Maskine Programmør Jensens pseudomaskine nr. 10.218.
Input-sprog: Sekvenser af instruktioner start-på-start, midt-på-start, slut-på-start, osv.
Hukommelse: Et højre-lager, et venstre-lager, samt en tælle-variabel.
Semantisk funktion: Instruktionerne udføres i rækkefølge, . . . osv.
eniksaM



Et »eksisterende« bibliotek af procedurer er også blot en abstrakt maskine. Et bibliotek med trigonometriske funktioner indeholder en funktion »sinus«, hvor brugerens (dvs. den programmør, som kalder den) forståelse er som følger »når jeg kalder sinus på et tal x, så får jeg som resultat en tilstrækkelig god tilnærmelse til den sande værdi af den matematiske sinus-funktion anvendt på x«. Der er en ganske god cost-benefit-ratio i denne anvendelse af abstrakte maskiner i forhold til, hvis vor programmør skulle ind og betragte en kryptisk algoritme for at beregne sinus, for hvert eneste kald af sinus-funktionen i sit program. Og ikke nok med det, han kan nu bruge sin generelle viden om trigonometriske funktioner til at ræsonnere om beregninger, som involverer adskillige funktionskald. Det er rimeligt overkommeligt at slå op i sin matematiske formelsamling og se, at de to beregninger »sinus(a)+cosinus(b)« og »2*sinus((a+b)/2)*cosinus((a-b)/2)« giver samme resultat.



2.4 Modeller for »hårde« maskiner

De abstrakte maskiner for Pascal, pseudokode og trigonometriske funktioner kunne vel siges at ligge ovre i »softwareafdelingen«. Abstrakte maskiner kan også smage mere af »rigtige« maskiner. Her følger et eksempel på en lille »maskin-maskine«, som har et rudimentært instruktionssæt, som arbejder på en hukommelse bestående af en regnestak og en tabel over variable og deres værdier. På trods af sin enkelhed, er det dog en maskine, som kan programmeres til at udføre en meget stor klasse af heltalsberegninger. Vi vil bruge denne maskine i de praktiske øvelser, så derfor beskrives den lidt mere detaljeret.

Maskine Mini-stak-og-variabel-maskine

Input-sprog: Sekvenser af instruktionerne

stak(tal), hvor *tal* er et heltal,
hent(variabel), hvor *variabel* er angivet ved et bogstav,
gem(variabel), — — —
udregn(operation), hvor *operation* er en af +, −, ⋆, >, >=, <, =<, =, =\=, /\, \/
hop(etikette), hvor *etikette* er et tal
n_ hop(etikette).

Instruktioner kan være forsynet med en etikette, og hver etikette kan kun forekomme som sådan én gang.

Hukommelse: En regnestak samt en lagercelle for hver mulig variabel.

Semantisk funktion: Operationerne udføres sekventielt, kun brudt i tilfælde af hop-instruktionerne. Betydningen af instruktionerne er som følger.

tak(tal), lægger *tal* på stakken,
hent(variabel), lægger den gældende værdi af *variabel* på stakken,
gem(variabel), fjerner en værdi fra stakken og lader den være den nye værdi af *variabel*,
udregn(operation), borttager de to øverste elementer fra stakken, udfører den specificerede operation på sædvanlig måde og lægger resultatet tilbage på stakken; sammenligningsoperationerne lægger specielle værdier »ja« eller »nej« på stakken,
hop(etikette), udførelsessekvensen brydes, og der fortsættes med den instruktion, som følger efter *etikette*,
n_ hop(etikette), fjerner et element fra stakken; hvis dette er »nej« udføres hop som ovenfor, ellers fortsættes med næste instruktion.
Udførelsen starter fra en hukommelse med tom stak og alle variable sat til 0.

eniksaM

(De lidt sjove symboler for operationer anvendt i *udregn* er valgt, så de ligner de tilsvarende Prolog-konstruktioner, som senere vil blive brugt til at modellere dem.)

For eksempel kan man overbevise sig om, at hvis maskinen gives nedenstående program, så vil den semantiske funktion generere en hukommelse, som indeholder en tom stak og variabelværdierne t = 0 og x = 161.

```
  stak(23)
  gem(t)
7 hent(x)
  stak(7)
  udregn(+)
  gem(x)
  hent(t)
  stak(1)
  udregn(-)
  gem(t)
  hent(t)
  stak(0)
  udregn(=)
  n_hop(7)
```

Opgave 2.2 På hvilke måder kan man forestille sig, at forfatteren kan have draget denne konklusion? Prøv iøvrigt at håndsimulere en del af udførelsen.

Vi ser altså, at vores forståelse af maskinen, som den er nedfældet ovenfor, gør det muligt at ræsonnere sig frem til resultatet af visse programudførelser. Men beskrivelsen siger ikke noget om, hvad der »sker«, hvis der f.eks. forsøges

hop til ikke-eksisterende etiketter eller at lægge »ja« og 7 sammen — det er ganske enkelt ikke defineret, disse detaljer interesserer os ikke. Igen det, at det er en abstrakt maskine. En hvilkensomhelst fysisk maskine, derimod, som på den ene eller anden måde simulerer den, vil fremvise en eller anden opførsel, f.eks. at den »går ned«.

Hvilke formål kan der mon ligge bag at beskrive denne maskine? Her er nogle muligheder.

- Den kan bruges i en bog om abstrakte maskiner som et eksempel på en abstrakt maskine.
- Den kan bruges i en lærebog om, hvordan »rigtige« datamaskiner ser ud — et forenklet eksempel i det første kapitel for at vise grundprincipperne.
- Den kan måske danne udgangspunkt for konstruktion af en ny og revolutionerende RISC-processor.
- Den kan bruges som et teoretisk instrument til at filosofere over, hvilke heltalsfunktioner, man kan udregne på en datamaskine. (Dvs. vor maskine skulle anses for at være en prototypisk repræsentant for begrebet »en datamaskine«).
- Man kunne implementere en fortolker for den, og så lade oversætterprogrammer generere programmer til den.

Opgave 2.3 Slå op i stikordsregistret til f.eks. Tanenbaums bog (Tanenbaum, 1999) og find ud af, hvad det ord, RISC, vi brugte ovenfor, dækker.

2.5 Virtuelle versus abstrakte maskiner

Tanenbaum (1999) er en meget velskrevet lærebog, som giver en sammenhængende fremstilling af fænomenet den moderne datamaskine. Vi får afdækket, hvad et operativsystem er, hvilke kredsløb maskinen består af, hvordan maskininstruktioner ser ud, osv. Tanenbaum viser os, hvordan et mikroprogram gør det muligt for kredsløbene at udføre maskininstruktionerne.

Tanenbaums centrale begreb er den *virtuelle* maskine. Han anskuer datamaskinen (med samt dens basale programmel) som en lagdelt konstruktion, hvert lag er en virtuel maskine, som er implementeret vha. en lavereliggende, virtuel maskine. Hvis vi slår op i en ordbog, vil vi se, at »virtuel« betyder noget i retning af »tilsyneladende« eller »faktuel« — i modsætning til »ægte

definitivt, virkelig«. Når Tanenbaum taler om en virtuel maskine, så er der tale om en abstrakt maskine, den har sit karakteristiske sprog, og der ligger en semantisk funktion begravet et eller andet sted i den.

Når vi sammenholder Tanenbaums virtuelle maskiner med de abstrakte maskiner, vi har indført, så bemærker vi, at Tanenbaums formål udelukkende er at være beskrivende. Det er et helt bestemt fænomen, han ønsker at forklare for os — de abstrakte maskiner, han herved præsenterer for os, er nogle, som eksisterer den forstand, at man er i stand til at få udført programmer i de tilhørende sprog. Vel er et maskinsprog blot en abstraktion over noget materie, i hvilket der løber visse elektriske strømme — men den er dog ikke mere abstrakt, end at det giver god mening at putte maskin-instruktioner ind i den og *rent faktisk* få dem udført (eller, måske mere præcist, på meningsfyldt måde opfatte en række fysiske begivenheder på denne måde). Vores begreb af abstrakt maskine er på denne måde mere generelt anvendeligt — selvom i grundsubstansen, de to begreber er identiske.

2.6 Sammenfatning

En vigtig konklusion på dette kapitel er, at en abstrakt maskine er en mental konstruktion, hvis formål er at indfange det essentielle i en instans af noget, vi forstår som en datamaskine — eller noget som »vi (i princippet) kan gøre med en datamaskine«. Det er det samme, det handler om, nemlig en indretning med et sprog bestående af nogle instruktioner og en semantisk funktion, som bestemmer betydningen af disse instruktioner. Hvorvidt denne er simuleret af et stykke elektronik, et program, håndsimuleret på papir — eller slet ikke simuleret — det betyder intet. Vi vil derfor udvide Tanenbaums vise ord

»Hardware and software are logically equivalent«

til følgende

»Hardware, software, AND THINK-WARE are logically equivalent«.

Opgave 2.4 Skitsér en Java-klasse svarende til den abstrakte maskine Summations-apparat. Overvej sammenhæng mellem de to begreber, klasse og abstrakt maskine.

Opgave 2.5 Beskriv et eller andet som en abstrakt maskine og overvej, hvilke fænomener, den kunne beskrive.

Opgave 2.6 Giv eksempler på datalogiske fænomener, som ikke på god måde kan forklares/forstås/beskrives ved abstrakte maskiner.

Opgave 2.7 Det går ud på at skrive et program til Mini-stak-og-variabel-maskine, som laver en heltalsdivision mellem to tal, placeret i variablene a og b, og efterlader resultatet i variabel c. Specificér først et program i et pseudosprog, f.eks. med while-løkker og andet, du måtte have brug for, og skriv det derefter om til den rigtige kode. Benyt den tilgængelige fortolker for Mini-stak-og-variabel-maskine til at afprøve løsningen. Start evt. med nogle simplere eksempler.

Opgave 2.8 Diskutér følgende påstande.

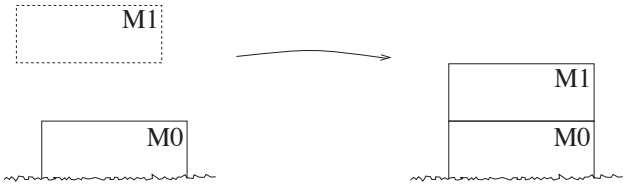
- En stak er en abstrakt maskine.
- En mængde er en abstrakt maskine.
- En abstrakt datatype er det samme som en abstrakt maskine.
- Abstrakte maskiner er smukke.
- Regnskabskontoret er en abstrakt maskine.
- Abstrakte maskiner kunne være opfundet af Aristoteles.
- Datalogi er læren om virtuelle maskiner.
- Datalogi er læren om abstrakte maskiner.

3 Implementation af abstrakte maskiner

Vi vil her forstå implementation i en passende abstrakt forstand. Ikke nødvendigvis, at der er tale om kørende implementationer, men at en maskines sprog og semantiske funktion i passende forstand realiseres i termer i en anden maskine.

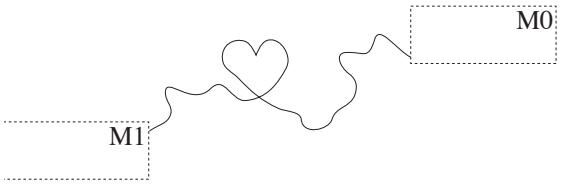
I den model, vi er ved at opstille for datalogiens fænomener, er de abstrakte maskiner de eneste objekter, som kan »foretage sig noget «. Så hvis vi ønsker at tale om implementation af en abstrakt maskine, må det være ved hjælp af andre abstrakte maskiner. Antag nu, at vi har implementeret en maskine M1 vha. en anden abstrakt maskine M0. Vi vil undersøge, hvad denne relation mellem M1 og M0 kan bruges til, inden vi begynder at se mere detaljeret på implementation.

- Rent praktisk. Vi kan opnå en fysisk realisering af M1 givet, at der findes en fysisk realisering af M0. Dvs. hvis vi har M0 kørende på en datamaskine på en eller anden måde — og vi tror på, at M0 fungerer korrekt, så kan vi forstå implementationen af M1 alene ud fra vores forståelse af M0.



- Teoretiske overvejelser. M1 er implementeret på M0, så det er muligt at få M0 til, på passende vis, at eftergøre de beregninger, som M1 kan foretage. Dvs. den udtrykskraft, vi har på M1 findes altså også på M0. Ønsker vi at ræsonnere om beregninger på M1, kunne vi måske ræsonnere i stedet om de tilsvarende beregninger på M0. Hvis yderligere, vi

kan implementere M0 på M1, så har vi vist, at de i passende forstand er ækvivalente. Dvs. teoretikeren kan argumentere snart udfra den ene og snart udfra den anden maskine — alt efter formål og bekvemmelighed.



– Definitorisk. Såfremt M0 har en veldefineret og præcis mening, så tjener en (beskrivelse af) en implementation af M1 på M0 som en definition af M1, dvs. hvad er dens karakteristiske sprog, hvordan er hukommelsen struktureret, og hvordan ser dens semantiske funktion ud. Hvis ydermere M0's sprog er af en passende støbning, kan en sådan implementation fremstå som en klar, letlæst og overskuelig definition af M1.

  Hvis yderligere M0 er realiseret som en kørende (dvs. virtuel) maskine, er en sådan definition *operationel*; måske ikke specielt effektivt eller med alle de finesser, man kunne forvente, men velegnet til testformål.

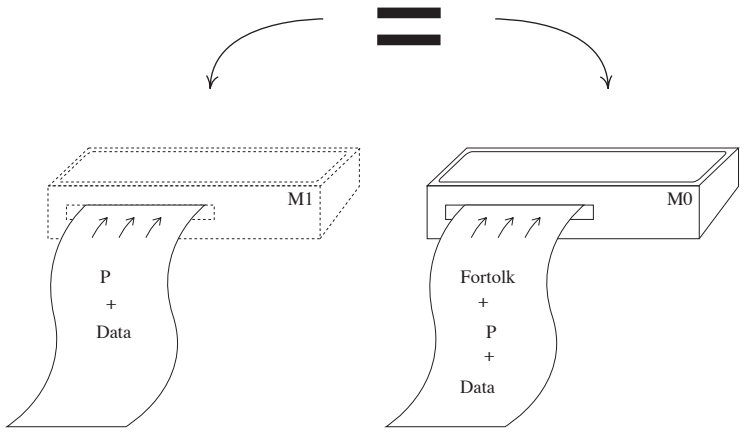
Vi vil vende tilbage til de to sidste pointer i et senere kapitler. Men vi kan allerede nu se nogle interessante aspekter ved den første pointe. Det giver anledning til en metodik for konstruktion af systemer. Den eller de personer, som konstruerer et kompliceret system, kan opdele systemet lagvis og koncentrere sig om at forbinde lagene ét ad gangen uden at skulle tvinges til at forstå alt på én gang — eller købe passende lag (programpakke, maskine, begge dele?) ude i byen og starte derfra.

3.1 Fortolkere

En fortolker for et sprog (eller en abstrakt maskine) er et program, som indlæser et program i sproget, som ønskes fortolket, og som så skridt for skridt »simulerer«, at det bliver udført. Vi vil definere det mere præcist i termer af abstrakte maskiner. — Vi benytter en praktisk notation for abstrakte maskiner nedenfor, hvor vi angiver den som et par af sprog og semantisk funktion, noteret ved hhv. S'er og F'er med subskript; hukommelsen er underforstået.

Definition. Lad M1=(S1, F1) og M0=(S0, F0) være abstrakte maskiner. *En M1-fortolker skrevet til M0* er et program i S0, lad os kalde det Fortolk, således at[1]
– for ethvert S1-program, P, og alle input, kaldet Data, gælder følgende,

F1(P+Data) = F0(Fortolk+P+Data).



Venstresiden angiver »det sande resultat«, højresiden det »faktisk beregnede«.

Lad os som eksempel overveje en fortolker for Mini-stak-og-variabel-maskine skrevet i Pascal. Denne maskine (dvs. dens sprog) indeholder ingen læseinstruktioner, så det objekt, som kaldes »Data«, kan vi se bort fra.

Vores fortolker er et Pascalprogram, som indlæser et Mini-stak-og-variabel-maskin-program, propper det ind i passende datastrukturer og simulerer en udførelse. Vi kan skitsere denne fortolker i pseudo-Pascal som følger.

```
program Mini-stak-og-variabel-maskin-fortolker (mini-program: text);
  passende konstanter og typer
  var mini-program-datastruktur: array[1..stor_nok] of ...
      program-tæller: ...
      stak: ..., lager: ...
begin
  læs indholdet af filen mini-program over i
    mini-program-datastruktur;
  program_tæller:= 1;
  while vi ikke er færdige do
  begin
```

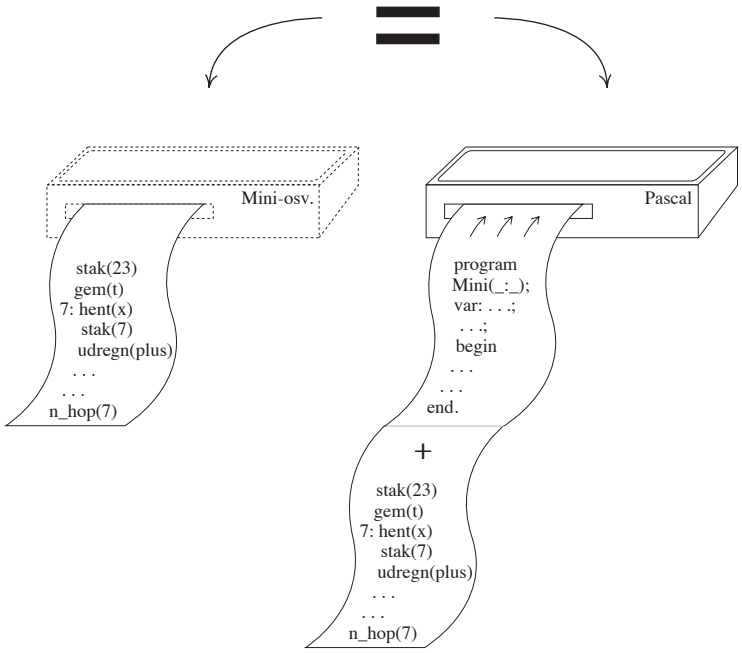
[1] Bemærk, at lighedstegnet i »ligningen« skal forstås med et gran salt. Var det rigtig matematik, ville det betyde, at de resulterende sluthukommelser skulle være ens. Men i praksis vil hukommelsen for M0 indeholde diverse hjælpevariable, og data er repræsenteret på en anden måde. Men alt dette har vi abstraheret væk. Matematisk orienterede læsere opfordres til at udtænke de ekstra symboler, som hører til en matematisk stringent formulering.

```
      aktuel_instruktion:= mini-program-datastruktur[program-tæller];
      case aktuel_instruktion of
        stak: tag argument og læg på stakken,
        o.s.v. — bemærk, at hopinstruktionerne piller ved
          program_tæller
      esac;
      program_tæller:= program_tæller + 1
    end
  end
```

Vender vi tilbage til definitionen, så er M0 altså Pascalmaskinen, M1 er Mini-stak-og-variabel-maskinen og Fortolk er programmet ovenfor. Betingelsen, for at der faktisk er tale om en fortolker, er som følger.

For ethvert program i Mini-stak-og-variabel-maskinsproget, P, gælder

Mini-stak-og-variabel-maskine-F(P) = Pascal(Fortolk+P).



Den generelle betingelse siger i dette tilfælde, at fortolkerprogrammet skitseret ovenfor, når det indlæser P, putter det ind sine datastrukturer, udfører instruktionerne én efter én som beskrevet i løkken. Den stak og det lager, som Mini-stak-og-variabel-maskinen ville have produceret, skulle så gerne kunne genfindes i fortolkerprogrammets datastrukturer. Og endelig, hvad vil det sige »at udføre fortolkerprogrammet«. Jo, det er bestemt ved den abstrakte maskine Pascal-maskine. Således kan vi forstå og analysere — og bruge —

fortolkerprogrammet uden at sætte os ind i, hvordan Pascaloversætteren er konstrueret.

Tilbage til metadiskussionen, hvad kan vi så bruge det her til? Jo, fortolkerprogrammet ovenfor kan gøre det muligt rent faktisk at køre programmer på Mini-stak-og-variabel-maskinen. For at tvære rundt i de flotte ord, kan man sige at vi ændrer Mini-stak-og-variabel-maskine fra at være rent abstrakt til at være en virtuel maskine. Hvis der afholdes afstemning om, hvorvidt maskinen eksisterer, vil nu ikke blot teoretikeren stemme for, men også praktikeren.

Skitsen af fortolkerprogrammet kan bruges som en generel skabelon for konstruktion af fortolkere for »maskinagtige« sprog. Hermed forstås, at programmer består af primitive operationer, som udføres sekventielt, evt. alterneret af hop-instruktioner.

Slutteligen skal vi ikke glemme at pointere, at fortolkeren kan bruges i en pædagogiske sammenhæng (ud over til at eksemplificere begrebet fortolker) til at forklare i en forenklet form, hvordan en datamaskine i princippet fungerer. Det ser man ofte gjort i lærebøger, og her fungerer den anvendte pseudo-Pascal eller -Java således som en abstraktion over den fysiske maskine på et relativt lavt niveau. (Se f.eks. Tanenbaum, 1999, s. 34).

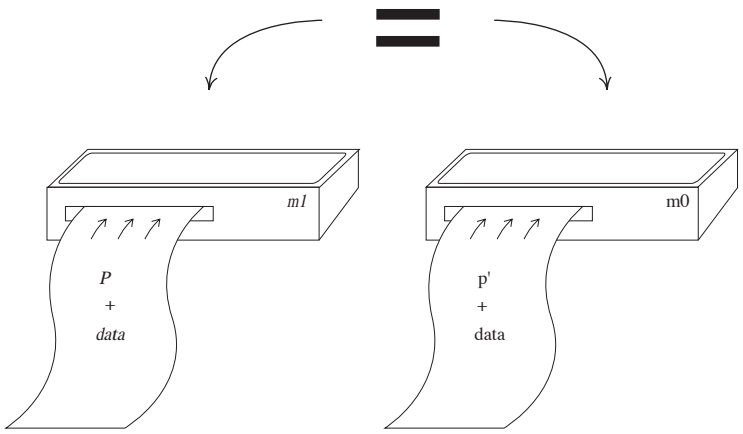
Fortolkeren beskrevet her tjener i princippet som en definition af Mini-stak-og-variabel-maskinen, men som sådan er den måske ikke idéel, da den er forfattet i et sprog med en relativt kompliceret semantik, og det kan derfor være svært at afgøre mere subtile spørgsmål. Vi viser senere en fortolker for Mini-stak-og-variabel-maskinen formuleret i Prolog, som er mere velegnet som sådan.

3.2 Oversættelse og oversættere

En anden måde at implementere en abstrakt maskine på, er gennem oversættelse. Antag, vi har en abstrakt maskine, M1, og et program P i M1's karakteristiske sprog. Princippet i oversættelse er at konstruere et andet program P′, til en anden abstrakt maskine M0, som har samme effekt som P, når det udføres.

Definition. Lad M1=(S1, F1) og M0=(S0, F0) være abstrakte maskiner og P og P′ programmer i S1, hhv. S0. P′ *er en oversættelse af* P såfremt, for alle input, Data, at

$$F1(P+Data) = F0(P'+Data).$$



Udvikling af programmer vha. pseudokode og trinvis forfining kan have karakter af oversættelse. Man skriver først et pseudoprogram, derefter erstatter man hver pseudoinstruktion med en »makro« af knapt-så-pseudo-instruktioner og lader de oprindelige pseudoinstruktioner stå som kommentarer i programmet.

Ordet oversættelse er måske oftere forbundet med programoversættere (eng.: compiler). En programoversætter, f.eks. fra Pascal til et maskinsprog, er en slags robot, som ved trinvis forfining konstruerer et udførbart maskinprogram ud fra et Pascalprogram (som er aldeles »pseudo« udfra et maskinsprogssynspunkt). Oversætteren er en korrekt oversætter, hvis det producerede maskinprogram har samme effekt, som Pascalprogrammet har i følge den abstrakte Pascalmaskine defineret ved referencemanualen for Pascal.

Programoversættere er dog en mere kompliceret affære end som just antydet. Det skyldes flere ting.

- Det er ikke nok at oversætte et program linie for linie som antydet ovenfor. Typedefinitioner og lignende har mere karakter af instruktioner

til oversætteren om, hvordan den skal oversætte de programlinier, som kommer senere.

– En direkte oversættelse af udtryk og sætninger vil resultere i langt fra optimal maskinkode. Der kan optimeres ved at huske på, hvilke mellemresultater, som vil komme til at ligge i hvilke registre osv., og der anvendes ikke blot ét, men flere forskellige mønstre for oversættelse af en given højniveaukonstruktion. (Der optimeres efter de faktiske argumenter, m.v.).

Opgave 3.1 Overvej, hvordan den dårlige oversætter og den gode oversætter hver især vil oversætte sætningerne

```
a[i]:= a[i] + 1 og a[i]:= a[i] * 2.
```

Samme spørgsmål for den sammensatte sætning

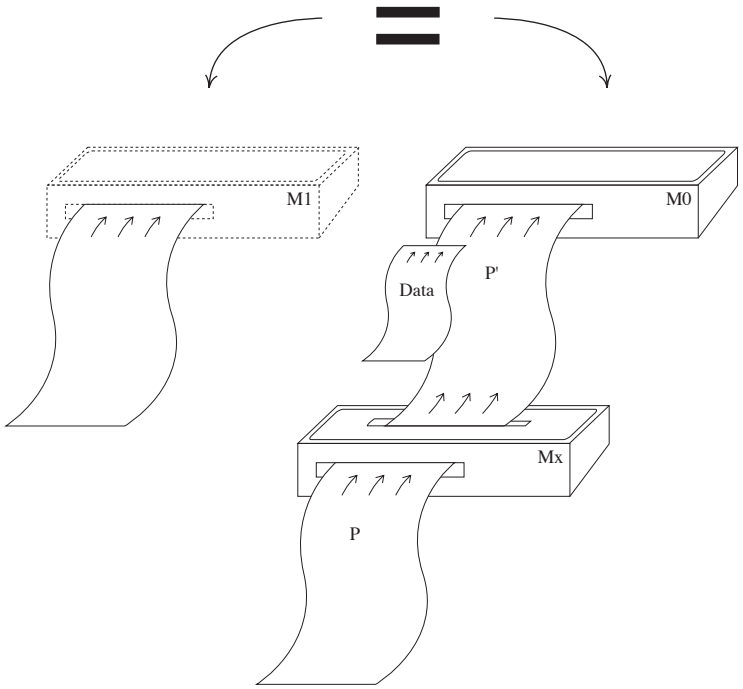
```
begin  a[i]:= a[i] + 1 ;  a[i + 1]:= a[i + 1] * 2  end.
```

Omkring programoversættere henvises til Aho, Sethi og Ullman (1986).

For god ordens skyld vil vi definere en oversætter (i modsætning til en oversættelse) indenfor vor ramme af abstrakte maskiner. En oversætter for, skal vi sige Pascal, kan forstås som en abstrakt maskine, hvis input-sprog også er Pascal. Den semantiske funktion producerer et nyt program i et andet sprog, f.eks. maskinsprog. Der skal så gælde, at Pascalprogrammet og maskinprogrammet har samme effekt, når de udføres på deres respektive maskiner.

Definition. Lad M1=(S1, F1) og M0=(S0, F0) være abstrakte maskiner. En *oversætter* fra M1 til M0 er en abstrakt maskine Mx=(S1,Fx), således at for alle S1-programmer P og input, Data, findes et S0-program P′ så

$$Fx(P)=P' \text{ og } F1(P+Data) = F0(P'+Data).$$



Der var altså en ganske abstrakt karakterisering af en programoversætter, blot som en abstrakt maskine. Men en oversætter er som oftest konstrueret ved at noget har skrevet et oversætterprogram, dvs. et program skrevet i et eller andet programmeringssprog, som indlæser et (Pascal-) program, analyserer det på passende vis og udskriver de tilsvarende maskininstruktioner. Oversætterprogrammer af denne art er også omfattet af vor ramme. Antag nu, at oversætteren er skrevet i et sprog Sy, som er defineret gennem en abstrakt maskine My. Ved elementær symbolmanipulation, ser vi, at dette oversætterprogram blot er en fortolker for Mx skrevet til My.

Opgave 3.2 Skitsér en passende tegning til at illustrere den sidste konklusion.

3.3 Abstraktionsmekanismer i programmeringssprog

De moderne programmeringssprog er gennemsyret af abstrakt-maskintankegangen. Hvis ikke vi havde abstraktionsmekanismer i programmeringssprogene, ville vi aldrig komme længere end til at skrive sorteringsprogrammer! Lad os betragte procedurebegrebet. Vi kan skrive et længere stykke kode, give det et navn P, og hver gang vi har brug for kodestykket, eller snarere det begreb, den implementerer, så refererer vi blot til navnet P. Der er jævnt godt

med, at programmeringssproget er blevet udvidet med en ny sætningsform, nemlig den med formen P.[2]

Skriver vi et programmodul med tre procedurer P1, P2 og P3, så svarer det til, at vi har implementeret en abstrakt maskine, hvis karakteristiske sprog indeholder de tre primitive instruktioner P1, P2 og P3. Hvis vi, inden vi faktiske kodede de tre procedurer, skrev den dokumentation, som specificerer deres effekt, så havde vi faktisk defineret en abstrakt maskine med karakteristisk sprog og semantisk funktion. — På tilsvarende måde kan vi også diskutere dataabstraktion, typer og klasser osv.

Opgave 3.3 Opstil en definition af, hvad der forstås ved en abstraktionsmekanisme. Konsultér eventuelt relevante fagleksika for at se deres definition.

Vi diskuterede trinvis forfining i afsnittet om oversættelse. Vi kan sige, at trinvis forfining baseret på programmeringssprogets abstraktionsmekanismer er smartere end blot at håndoversætte pseudoinstruktioner. Selve begrebshierarkiet (dvs. systemet af abstrakte maskiner implementeret i termer af hinanden) fremgår af selve programteksten, og pseudoprogrammerne bliver til »rigtige« programmer. Når vi så en gang imellem tyr til håndoversættelse, er det enten fordi den tilsvarende »rigtige« kode alligevel ikke fylder ret meget, eller fordi abstraktionsmekanismerne ikke har tilstrækkelig udtrykskraft.

Som et eksperiment vil vi implementere Mini-stak-og-variabel-maskinen gennem abstraktionsmekanismerne i et programmeringssprog. Dette programmeringsssprog kunne være Pascal, Java, C eller noget deromkring.

Først repræsenterer vi hukommelsen gennem passende variabelerklæringer. For at forenkle tingene, lader vi de specielle værdier »ja« og »nej« falde sammen med talværdierne 1 og 0. Det klarer vi med følgende konstantdefinitioner.

```
const ja = 1; nej = 0;
```

Stakken implementerer vi som følger.

```
var stak_indhold: array [0..stor_nok] of integer;
    stak_top: integer  := 0;
```

Hvordan disse variable bringes til at opføre sig som en stak, kan man læse om i enhver elementær lærebog i programmering. Lageret over variable kan repræsenteres på mange måder. Her vil jeg benytte Pascals symbolske »enumeration type«.

```
type  variabelnavn = [a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,æ,ø,å];
var   lager = array [variabelnavn] of integer;
```

[2] Denne idé er kørt til det ekstreme af bl.a. di Forini (1963), Christiansen (1985, 1990).

Endnu bedre ville det naturligvis være at implementere disse datastrukturer som pæne, abstrakte datatyper med veldefinerede tilgangsprocedurer.

Vi kan nu implementere i hvert fald nogle af instruktionerne i Mini-stak-og-variabel-maskinens karakteristiske sprog som procedurer, f.eks.

```
procedure stak(tal: integer);
begin
   stak_indhold[stak_top]:= tal;
   stak_top:= stak_top+1
end
```

Instruktionen udregn, der som argument tager en regneoperation, kræver lidt omtanke. Her må vi lave en erklæring, som repræsenterer regneoperationer.

```
type operation = [plus, minus, gange, større_end, osv.    ]
```

Nu kan vi repræsentere instruktionen udregn som følger.

```
procedure udregn(op: operation);
begin
   case op of
      plus: ...
      minus: ...
      ...
   esac
end
```

Det overlades til læseren at tænke sig til de udeladte detaljer. Nu kommer vi til et interessant punkt, og det handler om etiketter og hop-instruktioner. En etikette angiver her en position i et Mini-stak-og-variabel-maskine-program, og hop-instruktionen skal overføre kontrollen til den efterfølgende instruktion. Vi kunne forsøge at gå slavisk frem som ovenfor og definere en type af etiketter som værende heltal, men vi ville få svært ved at repræsentere det, at en etikette udpeger en instruktion i programmet. Vi kunne måske prøve at lave en procedure til at repræsentere positionen af en etikette. I såfald skulle vi skrive Mini-stak-og-variabel-maskine-programmerne á la følgende eksempel.

```
etikette(25);
stak(7);
hop(25)
```

Med et almindeligt kendskab til traditionelle programmeringssprog forekommer det ikke muligt at implementere procedurerne etikette og hop. Vi kan i stedet forsøge os med de etiketter, som findes i de fleste generelle, imperative programmeringssprog, dvs. at repræsentere etiketter i Mini-stak-og-variabel-maskine-sproget ved programmeringssprogets tilsvarende. Her kan vi passende benytte programmeringssprogets »go to«-sætning.

```
25: stak(7);
    goto 25;  /* hop(25) */
```

På denne måde fik vi almindeligt hop til at fungere. Det betingede hop er straks værre, fordi de fleste programmeringssprog er ret begrænsede, hvad angår at overføre etiketter som parametre. Der kan ligeledes være problemer med virkefeltsregler. Vi kunne forsøgsvis afprøve følgende.

```
procedure n_hop(e: label );
begin
   stak_top:= stak_top - 1;
   if stak_indhold[stak_top + 1] = nej then
      goto e
end ;
```

I nogle programmeringssprog ville dette her fungere og i andre ikke. Nogle sprog vil måske ikke tillade etiketter som parametre. Andre vil måske være uvillige til hop fra en procedure til en anden. I dette tilfælde er der kun et at gøre, nemlig hver gang, der er brug for en n_hop-instruktion, at gentage procedurekroppen ovenfor med den passende etikette substitueret ind (dvs. manuel oversættelse).

Antag nu for nemheds skyld, at vort programmeringssprog er tilstrækkeligt kapabelt, hvad angår behandling af etiketter, og at der er et modulbegreb, som tillader os at indkapsle de erklæringer, vi har skitseret ovenfor. Så bliver der tale om en nydelig implementation, som realiserer den abstrakte maskine Mini-stak-og-variabel-maskine. Vi kan da, hvis der laves et passende modul med navnet mini_stak_og_variabel_maskine, skrive programmer som følger.

```
use mini_stak_og_variabel_maskine;
begin
   stak(23);
   gem(t);
7: hent(x);
   stak(7);
   udregn(plus);
   gem(x);
   hent(t);
   stak(1);
   udregn(minus);
   gem(t);
   hent(t);
   stak(0);
   udregn(lig_med);
   n_hop(7)
end
```

3.4 Sammenfatning

Vi indførte i kapitel 1 begrebet af abstrakte maskiner og gav en række forskelligartede eksempler på datalogiske begreber, som kunne karakteriseres som abstrakte maskiner. I det nærværende kapitel beskrev vi, hvordan en abstrakt maskine kunne implementeres i termer af en anden abstrakt maskine. Vi beskrev tre måder at gøre dette på, fortolkning, oversættelse og vha. abstraktionsmekanismer i programmeringssprog. Disse begreber er også centrale i Tanenbaums bog (Tanenbaum, 1999).

Generelt handler implementation om at udtrykke en given maskine i termer af en måske lidt mere primitiv maskines sprog. På programmeringskurser, hvor man arbejder indenfor et enkelt programmeringssprog taler man om trinvis forfining, og det, vi kan kalde forfiningsstrukturen, fremgår gennem brug af abstraktionsmekanismer eller kommentarer i programmet. Når vi bygger større systemer i praksis, skriver vi sjældent ét stort program i et Pascalagtigt sprog. Vi kombinerer ikke blot selvkonstruerede programmoduler. Skal vi bruge noget, der har karakter af en database, ja så bruger vi da et databasesystem, kombinerer det med andre systemer osv., skriver forskellige programmer i forskellige programmeringssprog valgt efter, hvad de nu er gode til at udtrykke.

Den egentlige pointe er at påpege, at det er den samme metodik, man anvender, hvad enten man ønsker at forstå eller udvikle enkeltstående programmer, små systemer, mellemstore systemer eller megastore systemer. Eller det drejer sig om at forstå den basale organisering af en moderne datamaskine

— hvor Tanenbaums bog (Tanenbaum, 1999) er et ypperligt eksempel på en pædagogisk præsentation.

Slutteligen bør det dog påpeges, at kombinationen af abstrakte maskiner og implementation selvfølgelig ikke udgør en fuldstændig kalkyle for datamatiske systemer. En sådan skulle også rumme noget om at slå maskiner/sprog sammen. Hvor man kan kalde et system af abstrakte maskiner i lag-på-lag af implementationer en »vertikal« opdeling, er der også brug for en tilsvarende »horisontal« opdeling.

I øvrigt kan der henvises til en elegant, grafisk notation kaldet T-diagrammer, som er velegnet til at beskrive sammensatte systemer af sprog, oversættere og fortolkere (Earley, Sturgis, 1970).

4 Sprog og metasprog

Abstrakte maskiner blev præsenteret som et overbegreb, som indfanger forskellige lag af systemer, processorer og implementerede sprog, og hvor enhver abstrakt maskine har sit karakteristiske sprog. Sprog og maskine er duale begreber, maskinene definerer et sprog, og et sprog definerer en abstrakt maskine — i alt fald de fleste sprog, vi associerer med (tænkte eller »rigtige«) computere.

I dette kapitel tager vi udgangspunkt i sprog med henblik på at kunne karakterisere deres egenskaber. Vi interesserer os for et bredt udvalg af sprog, nogle med karakter af generelle eller specielle programmerings- (og grænseflade-) sprog, og desuden metasprog, som er beskrivelsessprog for sprog. Vi starter med at præcisere nogle helt generelle begreber om sprog.

Dette kapitel vil have karakter af en opremsning af begreber og beskrivelsesværktøjer, og resten af bogen vil så benytte disse og udfylde dem med substans.

4.1 En ontologi for sprog

Opgave 4.1 Slå op i et leksikon og find ud af, hvad ordet »ontologi« står for.

Datalogiens begreb af sprog er ikke sammenfaldende med det, lingvister kalder sprog, men der er mange analogier og lighedspunkter, som gør, at (meta-) sprogbrugen indenfor datalogi med fordel kan låne fra lingvistikken.

Vi vil skelne mellem et sprogs *syntaks* og *semantik*, og (ligesom lingvisterne fandt ud af på et tidspunkt) er det også nødvendigt at inddrage *pragmatik* for at få en dækkende forståelse af et sprog. Vi vil nu forklare mere indgående, hvad vi forstår ved disse egenskaber.

4.1.1 Syntaks

Et sprogs syntaks er de egenskaber, der handler om sprogets form. Prolog, for eksempel, har noget, der hedder klausuler, regler, mål osv., og disse skal skrive på en helt bestemt måde for at være »rigtige«. Den præcise syntaks har betydning, når vi skriver programmer, viser dem til hinanden og diskuterer frem og tilbage, hvorvidt der nu er brug for det ene eller det andet prædikat til et eller andet formål. Samtidig skal et fungerende Prologsystem (f.eks. baseret på en fortolker) kende denne syntaks, så det kan indlæse og repræsentere klausuler og programmer for at kunne behandle dem korrekt.

Vi vil betragte et sprog som et system af fraser: En *frase* repræsenterer et udsnit af en tekst,[1] vi vælger at betragte som en meningsfuld enhed, som vi kan tale om, klassificere og knytte betydninger til. Følgende tekstudsnit, for eksempel, genkender vi som en frase svarende til som et mål i Prolog med to argumenter.

```
p(X, a)
```

Et udsnit af en programtekst, som helt klart ikke repræsenterer en frase, får vi f.eks. ved at tage afslutningen af én regel og begyndelsen af den efterfølgende:

```
, Y). p(X, Y):- q(
```

Sprogets fraser klassificerer vi i et antal *syntaktiske kategorier* — i Prolog kan vi eksempelvis tale om programmer, klausuler, mål osv. Det svarer fuldstændigt til naturligt sprog, hvor vi f.eks. på dansk taler om hovedsætninger, bisætninger, udsagnsord, osv.

En syntaktisk kategori samler fraser eller former, som er beslægtede udfra deres funktion i sproget. Nye fraser kan bygges ved at vælge fraser af passende kategori og sætte dem sammen efter faste regler. Vi kan f.eks. bygge en klausul i Prolog ved at vælge et mål, som vi kan kalde dens hovede og et række af andre mål, som vi sætter sammen til dens krop.

Vi skelner mellem konkret og abstrakt syntaks, hvor den abstrakte syntaks fjerner tilknytningen til en bestemt konkret syntaks, hvor det sidste kan handle om konkrete tegnsekvenser eller grafiske fremstillinger.

4.1.2 Abstrakt syntaks

For aritmetiske udtryk beskriver den abstrakte syntaks et niveau, hvor vi har én syntaktisk kategori, *udtryk*, og et antal (abstrakte) regler for, hvordan udtryk kan bygges ud fra simplere udtryk. Vi kan eksempelvis sammenfatte en abstrakt syntaks på følgende måde.

[1] . . . en tekst eller en anden syntaktisk materie, f.eks. et struktureret billede på en skærm.

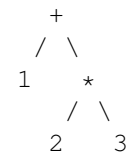
– Et *heltal* er et udtryk (hvor begrebet af heltal skal forstås i matematisk forstand).

– Et *plusudtryk* er sammensat af to udtryk.

– Et *gangeudtryk* er sammensat af to udtryk.

Et udtryk i den abstrakt-syntaktiske forstand er således en matematisk (eller, om man vil, rent mental) konstruktion, og ikke noget som vi kan tilskrive en bestemt fysisk eller tekstlig form. Tag for eksempel et udtryk, som er et plusudtryk sat sammen af to udtryk, den ene heltallet ét, det anden et gangeudtryk sat sammen af heltalsudtrykkene to og tre. Vi har alle fra folkeskolen en klar forestilling om, hvad det er for et udtryk, som er på tale, og vi ved også godt, at det kan skrives på mange forskellige måde, men uanset hvad, så er det tale om ét bestemt, veldefineret udtryk.

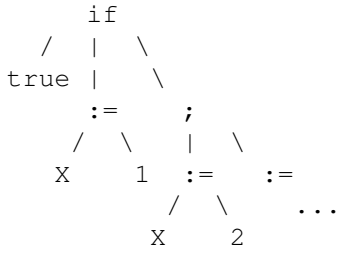
Vi bruger betegnelsen *abstrakt syntakstræ* for en sådan konstruktion, som vi så kan repræsentere på forskellig måde. Vi vil undlade at trætte læseren med matematiske formaliseringer af abstrakt syntaks.[2] Vi kan benytte en grafisk fremstilling af et abstrakt syntakstræ, hvor eksempelvis det føromtale udtryk kan præsenteres således:



Træets knuder er en symbolsk fremstilling af de enkelte operatorer eller regler (f.eks. for +), og deltræerne er abstrakte syntakstræer for deludtrykkene. Denne notation benyttes generelt, f.eks. kunne man forestille sig, at teksten

```
if true then X:= 1 else {X:= 2; Y:= 3}
```

hensigtsmæssigt kunne opfattes som følgende træ.



[2]De nysgerrige kan f.eks. se under universel, mange-sortet algebra, hvor abstrakte syntakstræer kan karakteriseres vha. ækvivalensklasser af initiale algebraer.

Fordelen ved at betragte fraser som abstrakte syntakstræer er, at det kun er den egentlige struktur, som er tilbage. Det er så at sige formen uden form, eller i alt fald ren-destilleret til det mindste som lige akkurat kan kaldes form; detaljer som vedrører den konkrete fremtoning — og som er grundlæggende ligegyldige for at knytte en semantik til en frase — er bortraderet.

Det samme abstrakte syntakstræ kan repræsentere forskellige konkrete fremtoninger på én gang. F.eks.

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

Af dette eksempler fremgår det, at vi i denne sammenhæng ikke finder parantessætning værdig til omtale på det abstrakt-syntaktiske niveau.

4.1.3 Konkret syntaks

Man kan i princippet godt undvære en tekstlig form for programmerings- og andre computersprog, ved f.eks. at benytte en grafisk grænseflade, hvor man bygger abstrakte syntakstræer ved at udvælge og ekspandere knuder i træet. Sådanne grænseflader findes faktisk, men som regel blandet med en tekstlig grænseflade. Et eksempel på en sådan blandet grænseflade er det kontrolvindue, som dukker op på skærmen, når man forsøger at gemme en fil på f.eks. en Macintosh eller Windows-maskine.

Ved konkret syntaks forstår vi et sprogs fysiske fremtræden i en eller anden form, f.eks. som en tekst. Vi vil her begrænse os til konkret syntaks i form af tekst, men det skal understreges, at de fleste begreber også er anvendelige i forhold til andre typer grænseflader. Traditionelt opdeles konkret syntaks i to niveauer,

- *leksikalsk syntaks*, som svarer til den konkrete udformning af et sprogs mindste dele, kaldet *leksikalske symboler*, f.eks. hvordan en variabel genkendes fra andre symboler eller hvordan reelle tal skal skrives.

- *strukturel syntaks*, som beskriver, hvordan de leksikalske symboler stykkes sammen til at danne fraser, som repræsenterer abstrakte syntakstræer.

Der er flere gode grunde til denne opdeling. Det giver en mere overskuelig beskrivelse af den strukturelle syntaks, som ydermere bliver uafhængig af, hvilket tegnsæt, som faktisk anvendes. Dvs. man kan benytte sammen strukturelle syntaks i en referencemanual, som dækker flere implementationer af

det samme sprog (som kan adskille sig ved om kodeord skal skrives med stort eller småt, normal eller fed skrift, hvis dette er muligt, om der nu kan stå æ-ø-å i variabelnavne eller ej osv.). Yderligere gælder det, at man kan benytte (og traditionelt benytter) forskellige beskrivelsesværktøjer og implementationsmetoder for de to niveauer.

Leksikalsk syntaks

Lad os diskutere konkret syntaks for sproget Prolog. Her kendes variable fra atomer ved, at de begynder med et stort bogstav. Der findes andre Prologversioner, hvor variable markeres ved at sætte en stjerne foran variablenavnet, og andre igen, hvor navne, som begynder med et af bogstaverne »`x`«, »`y`«, eller »`z`« er variable, alle andre atomer. I det sidste tilfælde er »`hest`« altså et atom, hvorimod »`zebra`« er en variabel. Hvorvidt vi bruger den ene eller anden konvention, er det dog stadig det samme sprog – det ser blot lidt forskelligt ud. Der findes også versioner, hvor man i regler skriver »`if`« i stedet for »`:-`« og »`and`« i stedet for »`,`«, men det ændrer jo heller ikke noget fundamentalt ved sproget, en regel er og bliver en regel!

Formen af de leksikalske symboler i et sprog kan beskrives ved *regulære udtryk*, som vi introducerer i det følgende. Endelige tilstandsmaskiner, som vi introducerer senere, er indretninger, som er velegnede til (abstrakt karakteristik af algoritmer til) genkendelse af regulære udtryk, men de kan også benyttes som et grafisk beskrivelsesværktøj for leksikalsk syntaks. Der kendes eksempler på såkaldte fjerde-generationssystemer, hvor udvikleren benytter en grafisk grænseflade til at tegne tilstandsmaskiner, som så definerer den leksikalske syntaks, som bliver en del af det nye system (som udvikleren er i gang med at udvikle).

Strukturel syntaks

Den strukturelle syntaks af et sprog angiver, hvorledes dets leksikalske symboler stykkes sammen til at repræsentere abstrakte syntakstræer. Ofte er den strukturelle syntaks (som er en del af den konkrete syntaks) væsentligt mere detaljeret end den tilsvarende abstrakte syntaks. Eksempelvis kan man klassificere udtryk udi primære, sekundære og tertiære udtryk med det formål at få beskrevet elimineret tvetydigheder, således at man får gjort klart at `1 + 2*3` ikke det samme som `(1+2) * 3`.

Vi kan definere et begreb af konkrete syntakstræer ud fra beskrivelser af leksikalsk og strukturel syntaks på analog måde svarende til abstrakte syntakstræer. Hvis der er tale et tekstbaseret, konkret syntaks, så vil den faktiske

tekststreng kunne aflæses ved at følge en passende vej ned gennem træet og opsamle de leksikalske symboler, der sidder som blade i træet.

For at beskrivelser af et sprogs syntaks skal kunne give god mening, må man forvente, at der er en entydig måde at afbilde konkrete syntakstræer over i abstrakte ditto (men sjældent entydigt den anden vej).[3]

Parantessætning i aritmetiske udtryk, som vi så hånligt bortviste fra den abstrakte syntaks domæne, må nødvendigvis karakteriseres på det strukturelt-syntaktiske niveau (ét sted må man nødvendigvis henføre paranteserne til!). Og der er en velkendt erfaring, at paranteser ikke blot er bekvemme, men også nødvendige, når vi noterer aritmetiske operatorer mellem deres argumenter og benytter sædvanlig prioritet mellem operatorerne. Uden paranteser ville det være umuligt at udpege det abstrakte syntakstræ, som er indeholdt i teksten `(1+2) * 3`.

I forbindelse med konkret syntaks omtales ofte et begreb kaldet *syntaktisk sukker*. Det er, når der indføres særligt bekvemme måder at skrive programtekst på (dvs. at udpege abstrakte syntakstræer gennem en konkret form), som ud fra et puristisk synspunkt kunne undværes. Eksempelvis kan man påstå, at infiks-peratorer med operator-præcedens er syntaktisk sukker. I Prolog, for eksempel, er der en grundlæggende syntaks for termer, så man kan skrive udtrykket fra før som `*(+(1,2),3)` eller generelt,

operator`(` *underterm*`,` *underterm*`,` ...`).`

Dette er i en eller anden filosofiske forstand smukt, det er en ren form, som giver en entydig sammenhæng mellem abstrakt og konkret syntaks.

4.1.4 Kontekstafhængig syntaks

De fleste programmeringssprog indeholder større eller mindre grad af kontekstafhængigheder, som ikke kommer med i den rent træstrukturerede model. Eksempelvis, at variable skal erklæres for at måtte kunne bruges, og at de har bestemte typer, og at nogle operatorer kan anvendes på nogle typer af udtryk og ikke på andre — og at operatorerne betyde noget forskelligt, når man anvender dem på forskellige underudtryk. Vi kigger lidt mere på kontekstssensitiv syntaks i afsnit 4.6 om definite klausul-grammatikker. Kapitel 7 viser et eksempel på et sprog, hvor kontekstafhængig syntaks flyder sammen med semantikken, så man ikke rigtigt kan skille dem ad. Kapitel 8 beskriver kontekstafhængig syntaks i forbindelse med erklæringer af sædvanlige konstruktioner så som variable, procedurer og klasser.

[3] Vor matematisk orienterede læser kan fornøje sig med at præcisere denne betingelse i form af symboler.

4.1.5 Semantik

De egenskaber ved et sprog, der vedrører betydningen af dets syntaktiske konstruktioner, betegner vi sprogets *semantik*. Når vi taler om semantik, er der to aspekter: hvordan vi som brugere af sproget forstår det, og hvordan det er realiseret ved hjælp af en datamaskine. Disse to opfattelser skulle helst stemme nogenlunde overens, omend vi mennesker, dovne som vi er, har en tendens til at se bort fra kedelige og subtile detaljer omkring den maskinelle udførelse.

Lad os eksempelvis se på udtrykket »`A + B`« i et traditionelt programmeringssprog, Java, for eksempel; hvad får vi ud af det? Joh, at »`A`« og »`B`« lægges sammen, og at udtrykket repræsenterer deres fælles sum, og færdig med det. Men det er jo ikke hele sandheden, hvis nu tallene er meget store eller meget små, hvad så? For at afgøre det, må vi vide hvor mange bits, der er sat af til heltalsrepræsentation, og i grænsetilfælde, om maskinen bruger 1- eller 2-komplement-repræsentation. Og hvis »`A`« og »`B`« er reelle tal, eller værre, blot den ene af dem, så bliver det først kompliceret. Men til al praktisk brug handler »`A + B`« om at lægge to tal sammen, de mere komplicerede begrebsapparater kører vi kun i stilling i meget kritiske situationer.

For sproget Prolog er denne skelnen tydeligere — eller i alt fald for en delmængde af Prolog. Her kan vi præcisere en *deklarativ semantik* baseret på simple, matematisk-logiske begreber, og denne semantik udsiger, hvad der er et korrekt svar på en forespørgsel uden overhovedet at komme ind på, hvordan et sådan beregnes (eller endog, om det overhovedet kan beregnes). Tilsvarende har vi *operationel semantik*, som er en konkretisering af logisk resolution i form af en procedure, der beskriver en detaljeret udførelsesrækkefølge. Her kan man så formulere interessante spørgsmål om, hvorvidt en given operationel semantik er korrekt med hensyn til den deklarative, eller i hvor høj grad, den kan siges at være en tilnærmelse. I tilfældet Prolog kan vi anvende de matematisk-logiske begreber af *sundhed* og *fuldstændighed*, hvor en procedurel semantik er sund, hvis den kun producerer korrekte svar (dvs. som stemmer overens med den deklarative semantik), og den er fuldstændig, hvis den kan producere alle korrekte svar.

Opgave 4.2 (Kun for matematisk orienterede læsere). På hvilke måder adskiller en traditionel Prologimplementation sig fra idealerne om sundhed og fuldstændighed? Hvordan kan der, når der ses stort på den ultimative effektivitet, og i hvor høj grad kan dette forbedres? (Det kan være nyttigt at vende tilbage til denne opgave, når man har læst kapitel 10).

For andre sprog med en mere righoldig (læs: kompliceret) semantik, eksempelvis imperative for ikke at tale om objektorienterede sprog som Java, har

man ikke en tilsvarende simpel ideal-semantik at måle en konkret implementation (eller halv-abstrakt beskrivelse af en sådan) i forhold til, men man tilstræber selvfølgeligt at opnå noget, er ligner. Vi vender tilbage til dette, når vi skal diskutere konkrete metasprog.

4.1.6 Pragmatik

Pragmatik omhandler de egenskaber ved et sprog, som vedrører dets brug. Indenfor lingvistik går den pragmatiske skole ofte så langt, at den påstår, at sproget (her: naturligt forekommende menneskesprog) eksisterer og er defineret ene og alene i kraft af sin brug. Grunden til, at vi omtaler en hund som »en hund«, er, at du og jeg er enige om at en hund er »en hund«, og dette er socialt betinget, fordi vi har opnået succes (dvs. har opnået en ønsket kommunikation) ved at kalde en hund for »en hund«, og det fandt vi på fordi vi kopierede de andres sprog, da vi hørte dem kalde en hund for »en hund«.

For de sprog, vi arbejder med, ser situationen ved første øjekast lidt anderledes ud: Et sprog er defineret eksplicit ved en referencemanual eller lignende af nogle mennesker på et eller andet bestemt tidspunkt. Denne definition er objektiv, absolut og kanonisk og har eksistens uanset om folk gider skrive programmer i dette sprog eller ej.

Tager vi et større historisk perspektiv i øjesyn, minder det om billedet for naturligt sprog. Prøv at overveje, hvorfor en while-lykke angives med »`while`« i programmeringssproget Java. Det kunne eventuelt have noget at gøre med, at while-lykker i de tidligere sprog, som Java er inspireret af, også har heddet »`while`«, f.eks. Pascal, Ada, C, Simula og (mindst) tilbage til det banebrydende Algol60. Hvis man skal designe et nyt sprog, som har en konstruktion, som essentielt set er en while-løkke, så vil de fleste nok betegne konstruktionen med »`while`«, for så kan de kommende programmører nemt kunne genkende den, som det, den er, nemlig en while-løkke. Så programmeringssprogenes udformning er til en vis grad bestemt ved en social proces, der indgår i en historisk udvikling.

Vi benytter begrebet pragmatik her på en forholdsvis pragmatisk måde til at referere til interessante aspekter ved sprogenes brug og anvendelser, og som også har betydning for udformning af deres syntaks og semantik.

Uden at påstå vi nærmer os en fuldstændig ontologi for programmeringssprogs pragmatik, vil vi eksempelvis henregne følgende under pragmatiske aspekter.

- Hvilke anvendelsesområder er et sprog tænkt til? (Prolog er f.eks. velegnet til sprogbehandling og håbløs til matriksregning, og for Java er

det omvendt).

- Hvilke personer anvender sproget? Eksperimentelle fysikere? Professionelle programmører? Datalogiske forskere? Sproglige gymnasiaster? Databaseadministratorer? Tilfældige forbipasserede, som kun forventes at bruge et givet grænsefladesprog få gange i sit liv og gerne skulle kunne finde ud af at bruge det på få sekunder?[4]

- Hvor effektivt er sproget implementeret, kan det gøres mere effektivt eller ligger der iboende begrænsninger i sproget for, hvor effektivt det kan implementeres? (I generelle programmeringssprog lægges der ofte vægt på at opnå et udvalg af konstruktioner, som er et kompromis mellem effektivitet og elegant udtrykskraft. Sproget Pascal må ses som et ypperligt eksempel herpå, når man ser i forhold til, at det fremkom i starten af 1970erne.

- Omgivelserne for sproget. Hvilke værktøjer er til stede, er der f.eks. en editor, som kender sprogets syntaks? Integreret med en inkrementel compiler? Er der effektive undersøgelsesværktøjer så som tracere og debuggere? (Nogle sprog er nemmere at lave den slags ting for end andre. Basic er så banalt, at den slags nemt kan laves, og i den anden ende ligger objektorienterede sprog som Java, hvor der er iboende problemer med bl.a. dynamisk visualisering af datastrukturer.)

- Egenskaber som robusthed og sikkerhed — og i praksis ofte modsatrettede egenskaber som hurtig udvikling af programmer. Robusthed handler bl.a. om at et program skal kunne vedligeholdes i takt med at omverdenens krav ændres sig, og at dette kan ske »på en sikker måde« ved blot at ændre få ting lokalt. Sikkerhed handler bl.a. om, at der er en vis sikkerhed for, at et program, der ser ud til at virke korrekt, også har stor sandsynlighed for at fungere korrekt i ekstreme eller subtile situationer. (Typer og klasser forbindes almindeligvis med robusthed og sikkerhed, men ikke med hurtig udvikling; det typeløse Prolog har en høj udtrykskraft idet programmer med en »avanceret funktionalitet« kan implementeres på kort tid og med få programlinjer, men de andre egenskaber ligger det mere tungt med.)

- Valg af konkret syntaks, puristisk eller sukret? (Det mest puristiske sprog er vel nok LISP, hvor den konkrete syntax kan afbildes 1-1 til abstrakt syntaks og programmerne er stort set ulæselige, selv for den,

[4] Tænk på en brandalarm.

der har skrevet dem. I den modsatte ende af spektret finder vi Visual basic, eller det grafiske sprog i hvilken Legos robotter kan programmeres).

I praksis kan det være svært helt at skille syntaks, semantik og pragmatik fra hinanden, og her er typer et godt eksempel. Typer kan betragtes som noget rent syntaktisk, og typecheck blot som et udvidet syntakscheck; nogle forfattere omtaler typecheck som statisk semantik, og nogle varianter af typer nødvendiggør, at visse dele af typechecket må henlægges til udførelsestidspunktet.

4.1.7 Hvad forstås ved et metasprog?

Betegnelsen metasprog er også overtaget fra lingvistikken. Forstavelsen »meta« er græsk og betyder noget i retning af »efter« eller »bagefter«. I måden »meta« benyttes i forbindelse med sprog, skal det læses i retning af »om«, et metasprog er et sprog *om* et andet eller flere andre sprog. Traditionelt betegnes det sprog, som beskrives, objektsproget, vel en eksempel på dårligt oversat engelsk, »the object language« hvilket ret præcist betyder, det sprog, vi interesserer os vi (netop nu).[5] Et metaprogram er et program, hvis data repræsenterer programmer (eller andre udtryk) af et objektsprog, og (pr. definition) er programmer skrevet i et metasprog.

Det er altså *brugen* af et sprog, som gør det til et metasprog, og det, at der er tale om et eller andet bestemt objektsprog, er en abstraktion. (Meta-) programmøren *vælger* at betragte bestemt datastrukturer som repræsenterende det (abstrakte, virtuelle) objektsprog. Metaprogrammering er faktisk en af de ældste og mest udbredte forteelser indenfor datalogien, tænk blot på oversættere, fortolker, editorer og diverse programanalysatorer.

Metasprog rækker fra gammelkendte beskrivelsesformer for syntax, over diverse semantiske formalismer til specialiserede inputsprog, som anvendes i diverse afarter applikationsgeneratorer, compilergeneratorer og 4.-generationssystemer. Ofte vil denne type sprog tilstræbe at »ligne« nogle af de klassiske beskrivelsesformer, jvf. hvad vi diskuterede i afsnittet om pragmatik ovenfor. Eller metasproget kan være et generelt programmeringssprog, som qua sin anvendelse bliver et metasprog, og vi kan give os til at vurdere, hvor velegnet det er som sådant.

[5] Betegnelsen »objektprogram« (eng. »object program« eller »object code«) anvendes (uheldigvis) ofte for den maskinkode, som produceres af et traditionel programoversætter — et bedre betegnelse for dette er »target program/code/language«. Endelig skal det fremhæves, at dette her ikke har noget som helst med objektorienteret programmering at gøre!

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?

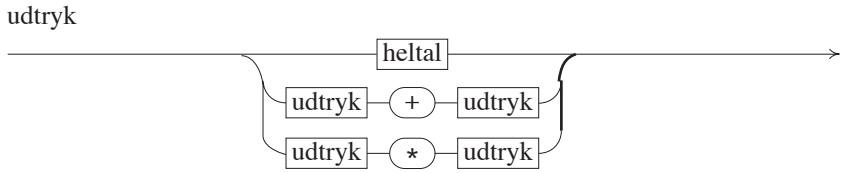
Opgave 4.4 Beskriv et regulært udtryk for tal i et programmeringsssprog, hvor følgende er eksempler på tal: 1, 3.14, -1.217e35.

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.

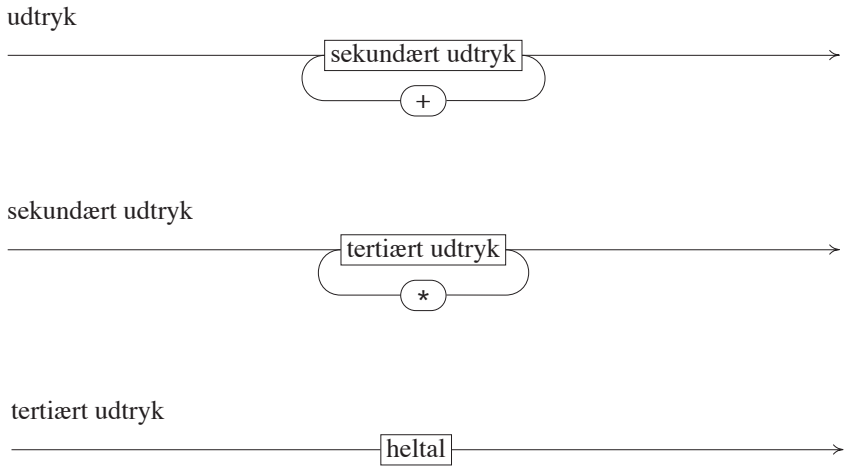
Vi vil diskutere dette i forhold til sprog af aritmetiske udtryk, hvis abstrakte syntaks, vi beskrev tidligere. Lad os antage passende leksikalske symboler defineret for sproget, og vi vil i første omgang skrive den abstrakte syntaks naivt om til et syntaksdiagram, som angiver en konkret, strukturel 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:

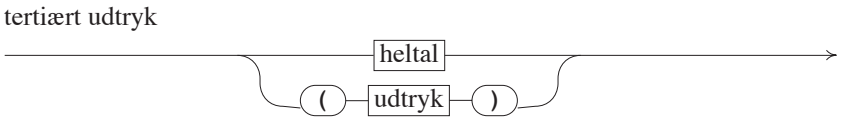
```
   *
  / \
 1   +
    / \
   2   3
```

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 parentes-

udtryk 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:

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

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.

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

4.2.3 Om metasprog for semantik

I en referencemanual beskrives semantikken for et sprog som regel i præcist naturligt sprog, hvor den på flere måde skelsættende Algol60-rapport (Naur et al, 1963) er et godt eksempel.

Der kan dog meget nemt opstå fejl og tvetydigheder i denne type beskrivelser, og derfor er der senere blevet udviklet forskellige matematiske formalismer til beskrivelse af programmeringssprogs semantik. Eksempelvis er der mange fejl og uklarhed i føromtalte Algol-rapport, når man går den efter. Vi skal undlade at trække læseren gennem en nærmere redegørelse for dette, men henvise til (Winskel, 1993), som er en velskrevet introduktion til de forskellige skoler indenfor formel semantik af programmeringssprog. Vi kan dog kort nævne de tre vigtigste retninger:

- Aksiomatisk semantik, også ofte kaldet Hoare-logik, som definerer semantik af algoritmiske sprog gennem bevissystemer baseret på pre- og post-betingelser.

- Denotationel semantik udviklet af bl.a. Scott og Strachey i 1970'erne. Hvert abstrakt syntakstræ afbildes homomorfisk (dvs. kompositionelt) over i et matematisk univers. For at kunne beskrive rekursion og ikke-terminerende programmer har det faktisk været nødvendigt at gentænke den klassiske mængdelære, og dette arbejde har bidraget til en øget teoretisk forståelse af automatiske beregninger og processer på niveau med Turings og Gödels arbejder i 1930erne.

- Plotkins operationelle semantik,[9], som ved deduktive regler, definerer sprogkonstruktionerne i forhold til en abstrakt maskine.

De fortolkere, vi præsenterer i Prolog senere i denne bog minder iøvrigt en hel del om Plotkins operationelle semantikker.

En automatiske compilergenerator, et 4.-generationsværktøj eller andre såkaldte applikationsgeneratorer har sit særlige metasprog for semantik, som kan være mere eller mindre slagkraftigt og mere eller mindre specifikt rettet mod en anvendelse. Systemet YACC (Johnson, 1975) tager som input en specifikation af en grammatik, hvori der er indflette stumper af C-kode, og YACC produceret så C-program, som foretager en syntaktisk analyse, og hvor de omtale stumper C-kode er indflettet på passende måde, så de aktiveres, hver gang

[9]Når vi i denne bog taler om en operationel semantik, refererer vi ikke til Plotkins arbejde, men bruger det i en bredere betydning for en semantik som enten kan udføres eller beskriver udførelse på en abstrakt måde.

der er blevet genkendt en delfrase af objektprogrammet. Det er så op til programmøren at vælge, om han vil udtrykke en fortolker, en oversætter eller noget helt tredje i disse kodestumper. Som vi har diskuteret det tidligere, kan en implementation (som defineret i kapitel 2) opfattes som en metasproglig definition af et sprogs semantik. En oversættelse fra abstrakte syntakstræer til et sprog, hvis semantik er kendt giver en entydig semantik. Tilsvarende med en fortolker.

Om et metasprog, må vi forvente, at det indeholde en måde at notere abstrakte syntakstræer og udvælgelse af deltræer. Ydermere må der forventes at være et tilpas slagkraftigt sæt konstruktioner, som er velegnet til at formulere semantik. Skal vi for eksempel beskrive semantikken af et sprog, hvori der indgår variable, er det formentligt praktisk at have datastrukturer til rådighed, hvor der kan defineres en association mellem variable og deres værdier.

Abstraktionsmekanismer i programmeringssprog har også en vis metasproglig karakter. Betragter vi eksempelvis en procedureerklæring for en procedure `P` med en parameter som er et heltal, kan man påstå, at procedurehovedet definere syntaksen for en ny konstruktion, ⟨*sætning*⟩::= `P (`⟨*heltal*⟩`)`, hvis semantik er bestemt ved procedurekroppen.

4.3 Objektorienterede sprog som beskrivelses- og implementationssprog

Objektorienterede programmeringssprog værdsættes almindeligvis pga. deres egenskaber i forhold til robusthed og deres værktøjer i form af klasser, nedarvning osv., som gør det muligt at definere datastrukturer og metoder til at manipulere dem med. For de fleste objektorienterede sprog er det muligt for en compiler at kontrollere, at der ikke foretages »uautoriserede« operationer på datastrukturer — samtidigt med, at klassedefinitionerne mere eller mindre kan læses som pæne matematiske definitioner af datastrukturerne.

Dette kan ses i modsætning til sprog uden typer, f.eks. Prolog, hvis egenskaber som metasprog, vi promoverer i de følgende afsnit. Her er det ganske enkelt *brugen* af Prologs termer, der definerer, hvad de repræsenterer, og det er så helt op til programmøren (eller lærebogsforfatteren) at opretholde konsistensen mellem denne brug og de ledsagende ekstra-programmeringssproglige beskrivelser i form af dokumentation eller lærebogsafsnit. Omvendt, så bliver objektorienterede programmer ofte lange og omstændige, netop fordi alle datastrukturer skal defineres i alle detaljer, før de kan benyttes.

Objektorienterede sprog som Java er på grund af denne robusthed velegnet som implemementationssprog i forhold til at programmere robuste over-

sættere og fortolkere, hvor Prolog på grund af sin meget kompakte udtrykskraft er mere velegnet til prototyper og illustrative formål (f.eks. lærebøger).

Udover at benytte Prolog som metasprog, bruger også en delmængde af det, kaldet Datalog, som et eksempelsprog, som implementeres vha. et objektorienteret sprog; i dette kapitel koncenterer vi os om repræsentation af syntaktiske begreber og vender tilbage til en fuld implementation af Datalog i kapitlerne 12–14, når vi har introduceret metoder til syntaksgenkendelse.

Som det objektorienteret programmerings- (implementations- og beskrivelses-) sprog benyttes det nu historisk interessante Simula, hvis syntaks for klassedefinitioner kan forekomme noget klarere end Javas. Omvendt, en læser, som er fortrolig med Java vil også kunne forstå eksemplerne formuleret i Simula; til sammenligning er den fulde implementation af Datalog gengivet i både Java og Simula i appendiks A og B.

Opgave 4.6 I de motiverende bemærkninger ovenfor blev der lagt stor vægt på betegnelserne »robust« og »robusthed«. Opstil en definition for, hvad disse betegnelser måtte betyde, og konsultér evt. litteratur om programmering og sammenlign.

4.3.1 Syntaks af eksempelsproget Datalog

Sproget Datalog er en delmængde af Prolog, hvor de ikke-logiske faciliteter er renset ud, og hvor argumenter er begrænset til atomer (dvs. konstantsymboler) og variable. Datalog er i øvrigt interessant som et databasesprog, da det udgør en generalisering i forhold til relationelle databaser (se kap. 9), samtidig med at det syntaktisk og begrebsmæssigt er en hel del enklere end traditionelle databasesprog. Datalog er ofte benyttet i forskningsartikler indenfor databaser, men har aldrig vundet indpas i praksis. I denne bog benytter vi det som et eksempel på et programmeringssprog, som vi, pga. dets enkelhed, kan beskrive en fuldstændig implementation af på forholdsvis overskuelig plads. For nærværende koncentrerer vi os om syntaks og syntaksbeskrivelse.

Datalog har følgende syntaktiske kategorier:

programmer, klausuler, mål, argumenter, atomer og *variable.*

Deb abstrakte syntaks for MiniProlog kan beskrives på følgende måde:

- Atomer og variable er to disjunkte og hver især uendelige mængder, og det eneste vi ved om dem er, at det giver god mening at tale om, hvorvidt to variable er ens, eller de ikke er det. Tilsvarende for atomer.

- Et argument er enten en variabel eller et atom.

- Et mål består at et atom (som vi kalder målets prædikatsymbol) samt en sekvens af et eller flere argumenter.

- En klausul består af et hoved, som er et mål, og en krop, som er en sekvens af nul eller flere mål.

- Et program er en sekvens af en eller flere klausuler.

Dette er en totalt immatriel beskrivelse af Datalog, men den er begrebsmæssig klar, således at vi verbalt over en telefonlinje ville kunne udveksle og diskutere Datalogprogrammer med en person på Mars (hvis ellers vedkommende og vi havde et fælles talesprog).

Den leksikalske syntaks for Datalog beskriver vi vha. regulære udtryk.

Vi benytter σ som forkortelse for udtrykket af små bogstaver, dvs. $\sigma = \mathtt{a} + \mathtt{b} + \cdots + \mathtt{å}$, Σ tilsvarende for store bogstaver og B for alle bogstaver, dvs. $B = \sigma + \Sigma$. Vi definerer følgende leksikalske symboler for Datalog ved hjælp af regulære udtryk.

atom-lex: $\sigma B*$

variabel-lex: $\Sigma B*$

kolon-streg: `:-`
(dvs. det regulære udtryk sammensat af tegnet kolon efterfulgt af tegnet streg, tilsammen beskrivende én bestemt tegnsekvens)

punktum: **.**

komma: **,**

start-parentes: **(**

slut-parentes: **)**

For at kunne opdele en tekst på entydig måde i en sekvens af leksikalske symboler må vi supplere beskrivelsen af Datalogs leksikalske syntaks med følgende konventioner:

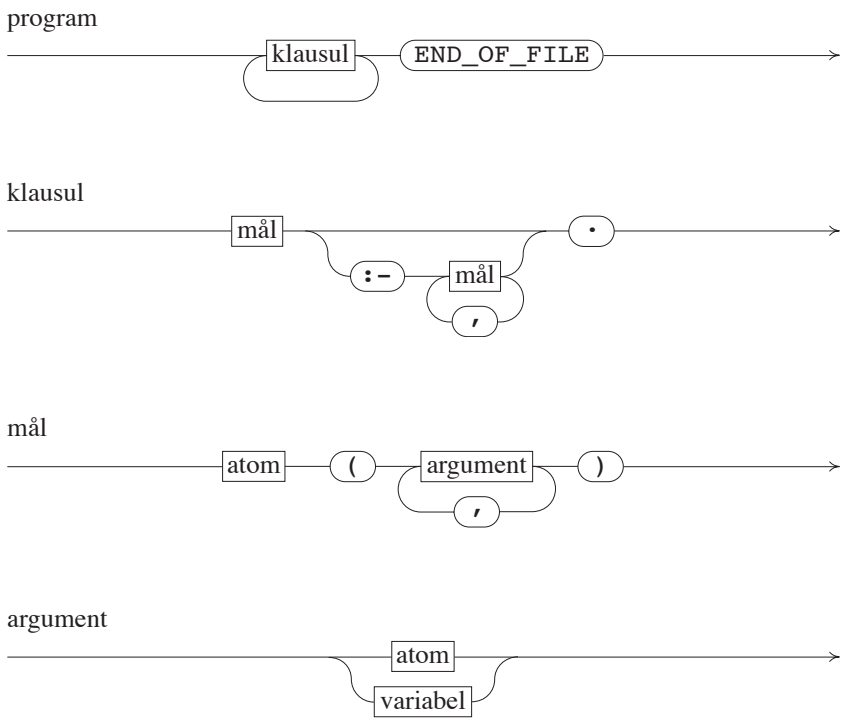
- Hvert atom og hver variabel udspænder en så stor delstreng som muligt,

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

Den tekstligt baserede konkrete syntaks for Datalog er nødvendig i forhold til at beskrive eller udvikle en traditionel implementation af sproget. I denne sammenhæng er det nyttigt at indføre et særligt leksikalsk symbol kaldet `END_OF_FILE`, som signalerer afslutningen af en tekstfil. Den viden, at en programtekst altid afsluttes med et bestemt symbol, er bekvemt i forhold til at formulere parsealgoritmer, som vi ser nærmere på i kapitel 13.

Ovenfor har vi været meget eksplicit med at navngive de leksikalske symboler, således at vi har dem til rådighed som byggesten, når vi kommer til den strukturelle syntaks. Af hensyn til læsbarheden vil man dog ofte i stedet for at skrive navnet på det leksikalske symbol, f.eks. »start-parentes«, benytte tekstrepræsentationen »`(`«, men man skal så være klar over at der er en bekvem skrivemåde og ikke andet.

Den strukturelle syntaks for Datalog er meget enkel, næsten for enkel i et pædagogisk sammenhæng, idet syntaksdiagrammer for den konkrete, strukturelle syntaks kan fås som en direkte omskrivning af den abstrakte syntaks, blot med indsættelse af nogle få terminalsymboler, men der er ikke behov for ekstra regler eller nonterminaler.



Der er ingen problemer her med præcedens eller associativitet, hvilket er yderst unormalt for et programmeringssprog.

4.3.2 Repræsentation af abstrakte syntakstræer

En abstrakt grammatik kan beskrives i et objektorienteret sprog på en ligefrem måde, hvor man benytter en samlende overklasse for abstrakte syntakstræer, som så specialiseres ud for de enkelte syntaktiske kategorier og videre for forskellige syntaksregler, når dette måtte være nødvendigt.

Vi viser den her i fuld tekst af hensyn til kommende eksempler. Vi kan ikke repræsentere sekvenser af delfraser direkte i Java eller Simula (der er ikke noget, der hedder »nul, en eller flere $\cdots$«), istedet benyttes rekursivt definerede datastrukturer og »hjælpeklasser«, som svarer til lister af mål og lister af argumenter. I tilfældet med et argument, som kan være *enten* en variabel *eller* et atom, benytter vi Simulas prefiksmekanisme. Som tidligere har vi samlet alle syntaktiske klasser under en fælles overklasse, kaldet `KATEGORI`: Her placeres de attributter eller egenskaber, som hører til det at være en syntaktisk kategori. — Herom senere.

Klassedefinitionerne, som er givet her, er taget direkte fra det endelige Datalogsystem; blot har vi udeladt attributter, som ikke er relevante på nuværende tidspunkt.

```
class KATEGORI;
  virtual: ...
begin end;


KATEGORI class PROGRAM;
begin ! En liste af KLAUSULer;
  ref(KLAUSUL) først;
  ref(PROGRAM) rest;
end;


KATEGORI class KLAUSUL;
begin
  ref(MÅL) hovede;
  ref(MÅL_LISTE) krop;
end;


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


KATEGORI class MÅL_LISTE;
begin
```

```
  ref(MÅL) først;
  ref(MÅL_LISTE) rest;
end;

KATEGORI class ARGUMENT;
begin end;

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

ARGUMENT class ATOM;
begin
  text id;
end;

ARGUMENT class VARIABEL;
begin
  text id;
end;
```

Brugen af tekststrenge til identifikation af atomer og variable forkommer en smule ad-hoc og er inspireret af en konkret syntaks. Det havde været mere stringent at benytte en abstrakt datatype, hvor man havde to operationer, $=$ og $\neq$. Tænker vi på klassedefinitionerne som en del af en implementation af sproget, er det hensigtsmæssigt at benytte en repræsentation, som gør implementerer disse to operationer effektivt, f.eks. ved at identificere hver variable og atom med et heltal, f.eks. bestemt ved den rækkefølge, de måtte optræde i en aktuel programtekst. Ud fra et effektivitetshensyn er brugen af tekststrenge meget uhensigtsmæssig.

4.3.3 Repræsentation af leksikalsk syntaks

På samme måde, som vi kunne beskrive abstrakte syntakstræer som datastrukturer i et objektorienteret programmeringssprog, kan vi lave klassedifinitioner svarende til de leksikalske symboler. Vi viser dette her, dels som en slags formel specifikation, og dels, så vi har dem, når vi senere kommer til beskrive algoritmer, som genkender leksikalsk analyse. Vi benytter stadig en notation, som minder om sproget Simula.

Vi definerer en generel klasse for leksikalske symboler og specialiserer ud til de enkelte slags symboler vha. prefiksmekanismen:

```
class LEKSIKALSK_SYMBOL;
begin end;

LEKSIKALSK_SYMBOL class ATOM_LEX(lex);
  text lex;
begin end;

LEKSIKALSK_SYMBOL class VARIABEL_LEX(lex);
  text lex;
begin end;

LEKSIKALSK_SYMBOL class KOLON_STREG;
begin end;

LEKSIKALSK_SYMBOL class KOMMA;
begin end;

LEKSIKALSK_SYMBOL class PUNKTUM;
begin end;

LEKSIKALSK_SYMBOL class START_PARENTES;
begin end;

LEKSIKALSK_SYMBOL class SLUT_PARENTES;
begin end;

LEKSIKALSK_SYMBOL class END_OF_FILE;
begin end;
```

De leksikalske symboler svarende til variable og atomer har en attribut, der identificerer dem udfra deres navne,[10] de andre underklasser af leksikalske symboler er uden attributter: Et komma er til enhver tid et komma og et punktum til enhver tid et punktum! Benyttes denne repræsentation af leksikalske symboler i et større program, hvor den f.eks. benyttes af en parser, kan der med fordel benytte Simulas `inspect-when`-konstruktion, eller tilsvarende i Java eller et anden sprog.

[10] Se diskussionen et par sider tilbage om det uhensigtsmæssige i at benytte en tekststreng til dette formål.

4.4 Prolog som metasprog

Prolog er et generelt programmeringssprog, som er særligt velegnet til at arbejde med symboler og (eksempelvis sproglige) strukturer, og som vi benytter i denne bog som et metasprog, bl.a. til at skrive fortolkere for andre sprog. I dette afsnit forklares, hvordan Prolog kan bruges som metasprog. Prolog indeholder også grundlæggende konstruktioner af metasproglige karakter, så et program kan udtale sig om sig selv og endog modificere sig selv.

4.4.1 Repræsentation af abstrakte syntakstræer

Termer i Prolog er velegnede som repræsentation af abstrakte syntakstræer. Det er ikke nødvendigt er erklære de indgående funktionssymboler,[11] de så at sige eksisterer alle fra starten. Her følger et par eksempler på Prologtermer, som man kunne forestille sig repræsenterende abstrakte sytakstræer.

```
+(1,*(2,3))
while(<(x,10), if(=(2,y), :=(x, +(x,1)), :=(y, -(y,x))))
```

Unification give en nem måde at (de-) komponere syntakstræer på. Hvis man unifier den sidste af de to termer ovenfor med `while(T1,T2,T3)`, får man netop udvalgt deltræerne.

Nogen gange kan også være praktisk at benytte en generisk repræsentation, hvor ethvert syntakstræ repræsenteres ved en struktur af formen `t(`*kategori*`,` *operator*`,`*liste-af-deltræer*`)`. Det første af de to træer vist ovenfor kommer da til at se sådan ud.

```
t(udtryk,+,[t(heltal,1,[]),
            t(udtryk,*,[t(heltal,2,[]),t(heltal,2,[])])])
```

Denne repræsentation fylder mere, er ikke særlig køn at se på, men til gengæld er den ofte være nemmere at programmere med.

Holder vi os til den simple repræsentation, har vi drage nytte af Prologs operatorbegreb (som man skal huske på er en rent syntaktisk tildragelse). Funktionssymbolerne svarende til de aritmetiske og logiske operatorer i eksemplet er allerede defineret som infiksoperatorer i de fleste Prologsystemer, og hvis vi selv tilføjer besværgelsen

```
:- op(800,xfx, :=).
```

kan termerne ovenfor skrives endnu mere bekvemt:

[11] I Prolog-litteraturen ofte kaldet funktorer.

```
1+2*3
while(x<10, if(2=y, x:= x+1, y:= y-x))
```

Abstrakte grammatikker kan formuleres som prædikater, der bestemmer mængder af (termer som repræsenterer) abstrakte syntakstræer. Mængden af abstrakte syntakstræer for de tilbagevendende aritmetiske udtryk kan defineres således:

```
udtryk(U1+U2):- udtryk(U1), udtryk(U2).
udtryk(U1*U2):- udtryk(U1), udtryk(U2).
udtryk(Tal):- integer(Tal).
```

Vi benyttede her Prologs indbyggede `integer` prædikat som et substitut for »de rigtige« heltal.

Metaprogrammer, dvs. programmer, der drejer sig om et eller andet objektsprog, vil ofte have en lignende struktur, hvor der blot er tilføjet ekstra argumenter, som rummer de resultater, som metaprogrammet har til formål at beregne. Der er i såfald tale om en *syntaksstyret proces*, hvilket vil sige, at et syntakstræ analyseres fra toppen og nedefter ved gennem en rekursiv dekomposition.

4.4.2 Fortolkere skrevet i Prolog

Prædikatet, som definerede abstrakte syntaks for aritmetiske udtryk ovenfor, kan udvides til en fortolker på følgende måde:

```
udtryk(U1+U2,V):-
    udtryk(U1,V1),
    udtryk(U2,V2),
    V is V1+V2.


udtryk(U1*U2,V):-
    udtryk(U1,V1),
    udtryk(U2,V2),
    V is V1*V2.


udtryk(Tal, Tal):- integer(Tal).
```

Formålet med fortolkeren er at udregne værdien af et udtryk, og denne værdi repræsenteres ved prædikatets ekstra argument. En regel for et sammensat udtryk, f.eks. `U1+U2`, kalder rekursivt prædikatet for de to deludtryk for at finde deres respektive værdier. Derefter udføres et stykke kode (skrevet i metasproget Prolog), som, givet deludtrykkenes værdier, bestemmer hele udtrykkets værdi.

Hver syntaktisk operator, f.eks. plusset i `U1+U2`, forsynes med en tilsvarende semantisk operation, i dette eksempel udtrykt ved Prologmålet

```
V is V1+V2.
```

Vi vil senere vise fortolkere for Mini-stak-og-variabel-maskinen og andre sprog, for derved at give en på én gang abstrakt og udførbar beskrivelse af deres semantik. Vi taler i denne situation som en *definerende fortolker*, og prædikater som

`udtryk(` *udtryk*`,` *værdi*`)`

omtales som *semantiske prædikater*. Generelt vil semantiske prædikater være af formen

syntaktisk-kategori`(`*abstrakt-syntakstræ*`,` *sem-arg*$_1$`, ...,` *sem-arg*$_n$`)`,

hvor argument *sem-arg*$_1$, ..., *sem-arg*$_n$ for givet abstrakt syntakstræ beskriver dets *semantiske relation*. I tilfældet med udtryk ovenfor falder dette ned til at være et enkelt datum, nemlig udtrykkets værdi, men ser vi på handlinger, som fører en før-tilstand over i en efter-tilstand, giver det god mening at tale om en semantisk relation. Man kunne eksempelvis forestille sig, at en definerende fortolker til en sætning `x:= x+1` tilknyttede en relation, som bl.a. indeholdt følgede elementer:

$$\begin{array}{rcl}
\langle \texttt{X=7} & , & \texttt{X=8} \rangle \\
\langle \texttt{X=666} & , & \texttt{X=667} \rangle \\
\langle \texttt{X=1, Y=32} & , & \texttt{X=2, Y=32} \rangle \\
\langle \texttt{A=17, X=1, Y=32} & , & \texttt{A=17, X=2, Y=32} \rangle
\end{array}$$

4.4.3 Oversættere skrevet i Prolog

Simple oversættelser kan specificeres meget lig fortolkeren ovenfor, her skal reglen blot i stedet beskrive, hvordan to deludtryks oversættelser sættes sammen til oversættelsen af det sammensatte udtryk.

Vi vil her benytte Mini-stak-og-variabel-maskinens sprog som målsprog for en oversættelse, og indføre nogle små hjælpeprædikater, som gør det nemmere er beskrive en oversættelse. Vi repræsenterer programmer til Mini-stak-og-variabel-maskinen som lister af instruktioner repræsenteret på en ligefrem måde, som vil fremgå af eksemplerne.

Hvis vi for eksempel ønsker at oversætte et plusudtryk, hvis to deltræer er oversat til henholdsvis `[stak(1)]` og `[stak(2),stak(3),udregn(*)]`, må vi forvente en samlet oversættelse

```
[stak(1), stak(2), stak(3), udregn(*), udregn(+)]
```

Denne sammensætning kan foretages ved to på hinanden følgende kald af `append`-prædikatet. Det vil visuelt fremstå ret kluntet, så derfor indfører vi et hjælpeprædikat, som vi noterer ved `<-`, som generaliserer `append` til at sammensætte et vilkårligt antal lister og enkeltelementer til en ny liste. Vi illustrerer semantikken[12] af denne nye konstruktion med eksemplet fra før. Følgende kald vil resultere i, at variablen `K` får den sammensatte liste vist ovenfor som værdi.

```
K <- [stak(1)]
      + [stak(2), stak(3), udregn(*)]
      + udregn(+).
```

Vi kan nu skrive en oversætter fra sproget af simple aritmetiske udtryk til Mini-stak-og-variabel-maskinen som følger.

```
udtryk(U1+U2,K):-
    udtryk(U1,K1),
    udtryk(U2,K2),
    K <- K1 + K2 + udregn(+).


udtryk(U1*U2,K):-
    udtryk(U1,K1),
    udtryk(U2,K2),
    K <- K1 + K2 + udregn(*).


udtryk(Tal, stak(Tal)):- integer(Tal).
```

Vi indfører nok et hjælpeprædikat til hjælp ved oversættelse, som benyttes til at generere nye etiketter, som vi får brug for til oversættelse af kontrolstrukturer. Betragt som eksempel sætningen `if x > 5 then y:= y + 1`. Den vil på fornuftig vis kunne oversættes til noget i stil med det følgende.

kode for `x > 5`
`n_hop(117)`
kode for `y:= y + 1`
`117`

[12] Bemærk sprogbrugen her. Vi har her opfattet definitionen af et prædikat som en udvidelse af sproget med en ny konstruktion. Vha. en operatordefinition giver vi den en passende syntaks og dens semantik er defineret ved nogle få linier Prolog. Bemærk også, at vi hermed har defineret en ny betydning for +, som er gyldig på højresiden af <−, hvor den fortolkes af de omtale få linier Prolog. Bemærk, at Prologs indbyggede `is` bestemmer en anden fortolkning af +.

Her benyttede vi etiketten `117`, men hvis der nu er flere `if`-sætninger i det program, som skal oversættes, kan vi ikke bruge `117` hver gang. Der skal en ny etikette til for hver eneste, `118`, `119`, `120`, osv. Derfor er der indført (dvs. forfatteren har programmeret) et prædikat, som hver gang det kaldes, vil producere en ny etikette. Betragt f.eks. dette lille kørselseksempel.

```
?-  ny_etikette(La), ny_etikette(Lb), ny_etikette(Lc)


La = 63,  Lb = 64,  Lc = 65
```

Hvis nu variablen `K1` er bundet til korrekt kode for betingelsen, og `K2` er bundet til korrekt kode for kroppen af `if`-sætningen, så vil følgende Prolog-besværgelser binde variablen `K` til korrekt kode for hele `if`-sætningen.

```
ny_etikette(Etikette),
K <- K1 + n_hop(Etikette) + K2 + Etikette
```

Opgave 4.7 Skriv definitioner i Prolog af en operator og prædikatdefinition, som implementerer konstruktionen, vi noterede med `<-`.

Opgave 4.8 Overvej, hvor stort et Java-program der skal til for at udføre den ovenfor beskrevne oversættelse fra aritmetiske udtryk til Mini-stak-og-variabel-maskinen.

Opgave 4.9 Skriv definition for prædikatet `ny_etikette`. NB: Måske du skulle vente, til du havde læst afsnit 4.5.4.

4.5 Prolog som metasprog for sig selv

Udover sin logiske kerne indeholder Prolog et udvalg af slagkraftige (og aldeles ekstralogiske) faciliteter af metasproglig karakter, hvor Prolog fungerer som metasprog for sig selv, således at programmer bl.a. kan inspicere og modificere sig selv.

Vi skitserer også, hvordan man på enkel måde kan skrive fortolkere for Prolog i Prolog, hvilket vi vil drage nytte af senere i kapitel 10.

4.5.1 Operatordefinitioner, syntaktisk metasprog

Prologs operatordefinition giver en meget elegant måde at beskrive systemer af operatorer med indbydes præcedens og associativitet, og er også velegnet til at forklare disse fænomener generelt. Operatordefinitioner er beskrevet så udmærket i Bratkos bog (Bratko, 1990), så det afstår vi fra her.

I Prolog er alle syntaktiske objekter termer, også klausuler og aritmetiske udtryk. Den regel, vi almindeligvis skriver

```
p(X,Y):- Y is X+1, q(Y).
```

under forudsætning af de sædvanlige prædefinerede operatorer, ville uden disse se sådan ud:

```
:-(p(X,Y), ','(is(Y,+(X,1), q(Y))).
```

Den sidste version er ikke for køn, men den viser alligevel noget interessant, nemlig at den grundlæggende abstrakte syntaks for Prolog er ualmindelig enkel, og at med en smule elegant syntaktisk sukker kommer det til at se ganske menneskeligt ud alligevel.

Effekten af at udføre en eller flere operatordefinitioner er, at man får modificeret den måde, Prologsystemet foretager sin indlæsning og udskrivning på.

4.5.2 Repræsentation af programtekst som termer

Vi har tidligere bemærket, at klausuler i Prolog blot er en slags termer, så hvis vi ønsker at skrive et Prologprogram, som arbejder med Prologklausuler, kan vi repræsentere en klausul ved sig selv. Et prædikat `omvend`, som laver en klausul `(X):- q(X)` om til `q(X):- p(X)`, kan programmeres på følgende måde:

```
omvend((Hovede:- Krop), (Krop:- Hovede)) .
```

Vi ser her specielt, at en Prologvariabel, f.eks. `X`, repræsenteres på metaniveauet som en variabel, nemlig sig selv. En sådan repræsentation kaldes en *ikke-grundet*[13] repræsentation. Dette virker umiddelbart fornuftigt og er i mange tilfælde bekvemt, når man skal programmere. Desværre medfører den ikke-grundede repræsentation ofte nogle semantiske problemer, som dukker op til overfladen i nogle centrale, indbyggede prædikater i Prolog, og som vi vil studere nærmere i det følgende.

Som et modstykke til den ikke-grundede repræsentation kan man foreslå en *grundet* repræsentation, hvor alt, incl. variable repræsenteres ved grundede termer, hvor man f.eks. kan anvende atomer af formen[14] `'*X'` til at repræsentere variable. Vor klausul fra før er da repræsenteret som `p('*X'):- q('*X').`[15]

[13] En grundet term i Prologterminologi er en term uden variable.

[14] Ifølge sædvanlig Prologterminologi såkaldt »quoted atoms«.

[15] Vi ser bort fra den præcise association mellem hvilke konstanter, der repræsenterer hvilke variable, det interessante er blot, at den samme variabel er repræsenteret på konsistent måde ved samme atom udi en hel klausul.

Fordelen ved den grundede repræsentation er, at man ikke er i tvivl om, hvordan ting skal forstås. Under antagelse af en grundet repræsentation er det klart, at `p(X)` repræsenterer en term, hvor argumentet endu ikke er kendt. Dvs. `X` fungerer som en *metavariabel*, i og med at den står i en position, hvor der forventes et stykke programtekst (mere præcist, en repræsentation af en term). Metavariablen `X` kan så senere i en proces instantieres til f.eks. atomet `a` eller angivelsen af en variabel `'*Z'`.

Ved den ikke-grundede repræsentation er det ikke klart, om `X` i `p(X)` fungere som et metavariabel eller den repræsenterer en objektvariabel. — Diskussionen kan forekomme subtil, da metasprog og objektsprog her er sammenfaldende, men det faktum, at Prolog benytter den ikke-grundede repræsentation til sine metasproglige konstruktioner medfører nogle problemer, som man ofte løber ind i ved praktisk programmering i Prolog.

4.5.3 Negation i Prolog er metasproglig

Negation i Prolog har ikke noget med negation i matematisk-logiske forstand at gøre, men findes som en metasproglig konstruktion kaldet på engelsk negation-as-failure, der er defineret på følgende måde.

```
\+ P:- P, !, fail.
\+ _.
```

Dvs. `\+ p(X)` fejler i det aktuelle program, såfremt `p(X)` stillet som forespørgsel til Prolog kan lykkes.

Vi kan illustrere problemet ved at antage, at programmet indholder ét faktum om `p`, nemlig `p(a)`. Betragt så de to forespørgsler

```
X=b, \+ p(X) og
\+ p(X), X=b.
```

I Prolog burde vi forvente, at resultatet var uafhængigt af rækkefølgen af de to delmål, men her kan man hurtigt indse, at den første forespørgsel give succes, mens den anden fejler. I den første kommer `X` i praksis til at fungere som en metavariabel, som medvirker til at bestemme det mål, som gives til `\+`, dvs. aktuelt `p(b)`. I den sidste bliver `X` opfattet som objektvariabel og delmålet `\+ p(X)` bliver tolket som »Der må ikke findes noget `X`, så `p(X)`«.

Med en grundet repræsentation kunne man undgå problemet med en vedtagelse af, at Prolog nægter at udføre et `\+` mål, som ikke er grundet, enten ved en fejlmeddelelse eller ved at henlægge et sådant mål indtil argumentet blev fuldt kendt (dvs. grundet på metaplanet). Den underliggende

fortolkning af `\+` må så indeholde en »oversættelse«, som indsætter »rigtige« variable. Under disse forudsætninger ville resultatet blive uafhængigt af udførelsesrækkefølgen. De to forespørgsler

```
X=b, \+ p('*X') og
\+ p('*X'), X=b
```

ville begge fejle, hvorimod

`X=b, \+ p(X)` og `\+ p(X), X=b`

giver succes. Dvs. brugen af den grundede repræsentation gav en måde at skelne de to situationer fra hinanden, således, at vi opnår en uafhængighed af udførelsesrækkefølgen.

4.5.4 Selvmodificerende programmer

Prolog indholder konstruktioner, med hvilke klausuler kan tilføjes eller slettes under udførelsen af et program. Udføres f.eks. kaldet `asserta(p(a))`, tilføjes faktummet til programmet som første klausul for prædikatet `p`. Der henvises til (Bratko, 1990) for en nærmere introduktion til `asserta` og dens makkere.

Igen her har vi et problem med den ikke-grundede repræsentation. Hvis nu der inde i en stor og kompliceret Prologklausul forekommer et kald af formen `asserta(p(X))`, så kan man ikke se, om resultatet under udførslen bliver tilføjelsen af en klausul, som siger at `p(X)` for alle `X`, eller at `p(`*værdi*`)`, hvor *værdi* er en konstant værdi, som `X` er blevet tildelt på et eller anden tidspunkt. Også her kunne man opnå en forbedret semantik ved at benytte en grundet repræsentation, så f.eks. `asserta(p('*X'))` tilføjede »`p(X)` for alle `X`«, hvorimod `asserta(p(X))` (`X` ikke grundet på udførelsestidspunktet) kunne resultere i en fejlmeddelelse. Det synes ikke rimeligt at kræve, at semantikken for `asserta` skulle være uafhængig af udførelsesrækkefølgen, således som vi ønskede det for negation-as-failure.

Der opstår yderligere et problem med `retract` (som sletter klausuler), såfremt der efterfølgende backtrackes ind i kroppen af en klausul, som er blevet slettet. Semantikken er grundlæggende ikke veldefineret, og i de fleste Prolog-systemer fremprovokerer fænomenet en kryptisk fejlmeddelelse fra det underliggende operativ system.

Man kan faktisk godt udforme konstruktioner, som minder om `assert` og `retract`, og som har en logiske tilfredsstillende semantik, ved at lade de specificerede tilføjelser (eller sletninger) være lokale i forhold til et enkelt delmål. Så man kunne f.eks. lade

```
q(Y) asserting p(a,'*X')
```

betyde, at `p(a,X)` var aktiv netop når `q(Y)` bliver udført, og hverken før eller efter.

Der findes adskillige forslag til revisioner af Prolog med udvidelser af denne art. Men omvendt, og så er vi ovre i pragmatikken, så mister vi nogle udtryksmuligheder ved at afskaffe de velkendte `assert` og `retract`. Eksempelvis er `assert` velegnet til *memoisering*, som i sin enkelthed går ud på, at når man har udført et prædikat én gang for et sæt værdier, så `assert`er man resultatet. Antag at prædikatet `p(X,Y)` benyttes til at beregne en funktion, som afbilder hvert `X` over i et entydigt `Y`, og at det er defineret ved `p(X,Y):- q(X,Y)`, hvor vi forestiller os, at `q(X,Y)` dækker over en meget tidkrævende beregning. Vi kan da tilføje memoisering ved at erstatte definitionen af denne her.

```
p(X,Y):- q(X,Y), asserta( (p(X,Y):- !)).
```

Det betyder således, at første gang `p(a,Z)` kaldes, og via kaldet til `q` resulterer i, skal vi sige `Z=b`, så tilføjes `p(a,b):- !` som et faktum om `p` som står før den generelle regel. Næste gang `p(a,A)` kaldes, fås `A=b` hurtigt fra dette faktum.

Det er også interessant, at `assert`ede klausuler overlever fra den ene forespørgel til den anden, så man kan benytte det dynamisk udviklende program som en hukommelse i implementation af et dialogsystem.

4.5.5 Prolog fortolket i Prolog

Som vi har antydet tidligere, er Prolog velegnet til at skrive fortolkere i, og man da også skrive fortolkere for Prolog i Prolog.

Denne lille fortolker er i litteraturen kendt under betegnelsen »the Vanilla interpreter«, men hvor den egentlig stammer fra, henligger i det uvisse.

```
solve(true).

solve((A,B)):-
    solve(A),
    solve(B).

solve(A):-
    clause(A,B),
    solve(B).
```

»clause« er et indbygget prædikat i de fleste Prologsystemer, som giver adgang til de klausuler, som ligger i databasen. Kaldet `clause(A,B)`[16] vil unificere `A` med hovedet af en regel og `B` til kroppen af samme klausul. Ved backtracking vil prædikatet generere alle klausuler, som matcher med det givne `A`. Såfremt der er tale om et faktum, instantieres `B` til »`true`«, som er et standardprædikat, som altid er opfyldt. Bemærk i tredje regel, at det aktuelle mål unifies med hovedet af den valgte klausul. De ekstra paranteser i anden regel indikerer, at der er tale om ét argument `(A,B)`, som er en struktur konstrueret ved funktoren komma.

Hvis Prologs database indeholdet et program om et prædikat `p(-)`, kan man enten spørge stille en forespørgsel `p(X)` direkte til Prologsystemet eller gennem fortolkeren som `solve(p(X))`.

Opgave 4.10 Vanillafortolkeren kan kun behandle en lille delmængde af alle Prologprogrammer. Lav en (ikke nødv. udtømmende) liste over faciliteter, som ikke er med, og giv evt. forslag til, hvordan de kunne bygges ind i Vanilla.

Man kan stille spørgsmålet, hvad man dog skal bruge Vanilla til, når den kun kan foretage sig noget, som systemet allerede kan i forvejen. Fidusen er, at Vanilla gør visse dele af udførelsen eksplicit, f.eks. valget af klausul, og det gør det således muligt at ændre lidt på den, ved eksempelvis at påvirke valg af klausul eller måske sætte en udskrift, som rapporterer dette til brugeren. Vi vender senere tilbage til Vanilla, når vi betragter programmeringsværktøjer som tracere og debuggere.

Man kan yderligere forfine Vanilla, så den også parameteriseres med det program, som er under udførelse, hvilket gør det hensigtsmæssigt at gå over til en grundet repræsentation af objektsproget. Herved får man det såkaldte `demo`-prædikat, som altså kalde således:

`demo(`*repr. af objektprogram*`,` *repr. af objektforesp.*`)`

Dette prædikat kan under særlige omstændigheder også bruges som programgenerator. En metavariabel i det første argument vil stå for en ukendt programstump, og metaniveauforespørgslen kan så give som svar et programfragment, som gør objektforespørgslen (dvs. den som er angivet som andet argument) beviselig. Dette emne ligger dog udenfor denne bogs domæne; specielt interesserede henvises til (Christiansen, 1998a, 1998b).

[16] I mange systemer er det nødvendigt at erklære et prædikat til at være dynamisk for at få adgang til dets klausuler vha. `clause`, eksempelvis `:- dynamic parent/2`. De kræves ofte også, at første argument i et kald af `clause` er instantieret tilstrækkeligt til at prædikatet kan genkendes.

4.6 »Meta« som generel programmeringsteknik

Den viden, vi har opnået om metafortolkere kan bruges som en generel og meget slagkraftig programmeringsteknik. Vanillafortolkeren, som den er beskrevet ovenfor, er tro mod Prologs semantik. Men vi behøver sådan set ikke respektere Prologs semantik, men kan tilføje alle mulige fiksfakserier, som behandler udvalgte prædikater på særlige måder.

Vi vil her vise, at denne teknik kan bruges til noget fornuftigt gennem et eksempel. Betragt følgende ganske almindelige Prologprogram, som handler om, hvordan man finder vej gennem nogle ensrettede gader, hvor vejkrydsende hedder `a`, `b`, `c`, `d` og `e`.

```
vej( a, b).
vej( b, c).
vej( c, d).
vej( d, e).


vej( X, Z):- vej( X, Y), vej( Y, Z).
```

Dette program er som sådant godt nok, men vi kan se at prædikatet vej (via sin sidste regel) er et eksempel på et transitivt prædikat. Transitivitet er en ofte forekommende egenskab ved prædikater af to argumenter, så det kunne være interessant, hvis vi i programmeringssproget kunne definere begrebet »transitivitet« en gang for alle. I såfald kunne man droppe den sidste klausul og blot erklære prædikatet som transitivt, og systemet skulle så indrette sin fortolkning derefter.

Det, at betegne et prædikat som *transitivt* er kun meningsfyldt på et meta-niveau, dvs. det kan ikke udtrykkes eller udnyttes direkte i programmeringssproget, idet vi ikke kan parameterisere over prædikatsymbolet. For at udtrykke det, vi gerne vil, må vi altså benytte Prologs evner som metaprogrammeringssprog, dvs. vi vælger en hensigtsmæssig repræsentation for vores objektprogrammer (som bl.a. handler om prædikatet `vej`), og bygger en passende metafortolker. En hensigtsmæssig repræsentation er her en, som tillader os at parameterisere over prædikatsymbolet. Det opnås ved at opskrive fakta om `vej`-prædikatet på følgende måde, idet vi definerer et nyt prædikat `solve`, nu med to argumenter, det første svarende til prædikatsymbolet og det andet svarende til argumentlisten.

```
solve( vej, [a,b]).
solve( vej, [b,c]).
solve( vej, [c,d]).
solve( vej, [d,e]).
```

Dvs. vi repræsenterer alle prædikater gennem et metaprædikat ved navn `solve`, hvor vi vel og mærket ikke har regler som svarer til indmaden af Vanilla. Vi kan nu programmere *om* prædikatet `vej`, f.eks. ved at skrive et faktum som følger.

```
transitivt(vej).
```

Betydningen af transitivitet, dvs. hvordan der skal tages hensyn til det under fortolkningen, kan programmeres ved følgende klausul.

```
solve( P, [X,Z]):-
    transitivt(P),
    solve( P, [X,Y]),
    solve( P, [Y,Z]).
```

Fordelen er nu, at vi kan benytte transitivitetsegenskaben i nye prædikater. Antag f.eks. at der findes et `forælder`-prædikat, som er defineret vha. en række fakta om vort metaprædikat, f.eks.:

```
solve( forælder, [margrethe, joakim]).
```

Vi kan nu definere et ane-prædikat som følger.

```
transitivt(ane).


solve( ane, L):-  solve( forælder, L).
```

På tilsvarende måde som vi definerede begrebet transitivitet, kan vi fortsætte med at definere et helt katalog af tilsvarende egenskaber, refleksivitet, symmetri og antisymmetri osv. osv. Resultatet bliver en omgivelse, hvor man programmerer på et højere plan, definerer nye prædikater gennem deres overordnede egenskaber i stedet for at skrive detaljerede klausuler.

Vi kan også generalisere teknikken yderligere ved at tillade strukturer på prædikatets plads, mod at vi så skriver tilsvarende klausuler for metaprædikatet `solve`. Der er således ikke blot tale om en hensigtsmæssig måde at strukturere Prologprogrammer på, men om at vi udvider sprogets udtryksskraft. Hvor prædikater (eller relationer) før var angivet ved konstantsymboler (f.eks. `vej`), tillader vi nu også udtryk, som evaluerer til prædikater (eller relationer).

Vi kan for eksempel implementere negation som en operator på prædikater, vi bruger det sædvanlige præfiks-minus, og implementerer det ved Prologs negation-as-failure, her noteret ved `\+`.

```
solve( -P, L):-  \+ solve( P, L).
```

Vi kan fortsætte medindføre en plusoperator på prædikater, hvis semantik er defineret på denne måde:

```
solve( P+Q, L):-  solve(P, L) ; solve(Q,L).
```

Osv. osv.

Opgave 4.11 En egenskab, som matematikere ofte postulerer om binære relationer er *symmetri*. En relation R er symmetrisk såfremt $\forall x, y : (x\,R\,y \Leftrightarrow y\,R\,x)$. Den naive Prolog-metaprogrammør, vil straks sige, at det kan vi snildt modellere sådan her.

```
solve( P, [X,Y]):-
    symmetrisk(P),
    solve( P, [Y,X]).
```

Men ak, man behøver ikke engang testkørsler eller Turingmaskiner sat i sving for at se at denne `solve`-regel uvægerligt vil give anledning til uendelige løkker. Vel ved vi fra kapitlet om Turingmaskiner, at man ikke kan konstruere en generel løkkechecker, som virker hver gang, men det burde være muligt at lave nogle tilføjelser til `solve`, som undgår nogle af de mest oplagte løkker. Kom med forslag og implementér dem eventuelt.

I øvrigt er det konkrete eksempel i dette afsnit inspireret af sproget Reflective Prolog (Barklund *et al*, 1994).

4.7 Definite klausulgrammatikker

Definite klausulgrammatikker (DCG) er en grammatiknotation, som indbygget i de fleste Prologsystemer. Oversættelsen fra grammatikregler til udførbare Prologregler er meget enkel. Grammatikkens nonterminalerne oversættes til mål, som har nogle ekstra argumenter, som har at gøre med den tekststreng, som analyseres. Grammatiknotationen er altså en syntaktisk sødet udgave af en bestemt form for Prologklausuler. Ikke desto mindre har grammatikkerne en særlig appeal. Det er nemt at konstruere »morsomme« eller interessante eksempler. For en teknisk introduktion af grammatikkerne og deres implementation kan henvises til de fleste lærebøger som Prolog, f.eks. (Bratko, 1990).

Vi viser her nogle eksempler på DCG. For at tydeliggøre grammatikkerne rent teknisk, viser vi stærk forenklede grammatikker for naturlig sprog. Det skal dog pointeres, at DCG er bedst egnet til prototype-formål og i pædagogiske sammenhænge — til en »rigtig oversætter« vil man betjene sig af andre metoder og værktøjer. Eksemplerne viser også, hvordan DCG gør det

nemt at beskrive kontekstsafhængig syntaks. DCG svarer til de såkaldte attributgrammatikker, undviklet af (Knuth, 1968) og beskrevet også i (Aho, Sethi, Ullman, 1986), som er *den* klassiske formalisme til beskrivelse af kontekstafhængig syntaks for programmeringssprog. Den notation, som DCG stiller til rådighed er dog væsentlig mere elegant end den, som benyttes i de omtalte kilder.

4.7.1 DCG som metaprog for forenklet naturligt sprog

Følgende grammatik er en omskrivning af en kontekstfri grammatik for en meget lille delmængde af det engelske sprog. Der tages ikke her hensyn til ental/flertal o.lign. — Grammatikreglerne kan skrives direkte som del af en Prologkildetekst, som i øvrigt også kan indeholde »almindelige« klausuler. Prologsystemet besørger af sig selv den nødvendige oversættelse af DCG-reglerne til tilsvarende Prologregler.

```
sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun.

verb_phrase --> verb.

verb_phrase --> verb, noun_phrase.

determiner --> [the].

noun --> [man].
noun --> [men].
noun --> [woman].
noun --> [women].
noun --> [apple].
noun --> [apples].

verb --> [eats].
verb --> [eat].
verb --> [loves].
verb --> [love].
verb --> [sings].
verb --> [sing].
```

En sådan grammatik kan benyttes til at afgøre hvorvidt givne lister af symboler tilhører det beskrevne sprog eller ej. Det kan f.eks. foregå ved brug af

et indbygget prædikat `phrase` på følgende måde (dette kan dog afvige i forskellige Prologversioner).

```
?- phrase( sentence, [the,man,eats,the,apples])

yes

?- phrase( sentence, [the,smølf,smølfs,the,smølf])

no
```

Formålet med `phrase` er at foretage et kald af *prædikatet* `sentence`, som har de rette argumenter, dvs. svarende til den måde grammatikreglerne blev oversat til Prolog.

Som sagt tages der ikke i denne grammatik hensyn til aspekterne omkring ental/flertal, hvorfor følgende »forkerte« sætning bliver godkendt.

```
?- phrase( sentence, [the,women,eats,the,apple])

yes
```

Da der heller ikke er nogen semantik koblet på, accepteres også »egentligt sludder« som det følgende.

```
?- phrase( sentence, [the,man,sings,the,apple])

yes
```

Det er iøvrigt interessant at benytte `listing`-prædikat for at se, hvilke Prologregler, som er kommet ud af vores grammatikregler.

```
?- listing(noun_phrase)

     noun_phrase(_1, _2) :-
               determiner(_1, _3),
               noun(_3, _2).
```

Information om ental/flertal for de enkelte dele af en sætning kan indkodes ved at tilføje argumenter eller attributter til nonterminalerne. De skrives på præcis samme måde – og fungerer på samme måde – som argumenter til Prolog-prædikater.

```
sentence --> noun_phrase(Number), verb_phrase(Number).

noun_phrase(Number) --> determiner(Number), noun(Number).

verb_phrase(Number) --> verb(Number).

verb_phrase(Number) --> verb(Number), noun_phrase(_).

determiner(_) --> [the].

noun(singular) --> [man].
noun(plural)   --> [men].
noun(singular) --> [woman].
noun(plural)   --> [women].
noun(singular) --> [apple].
noun(plural)   --> [apples].

verb(singular) --> [eats].
verb(plural)   --> [eat].
verb(singular) --> [loves].
verb(plural)   --> [love].
verb(singular) --> [sings].
verb(plural)   --> [sing].
```

Eksempler på forespørgsler:

```
?- phrase( sentence, [the,man,eats,the,apple])

yes

?- phrase( sentence, [the,man,eat,the,apple])

no

?- phrase( sentence, [the,man,sings,the,apple])

yes
```

Vi kan studere, hvorledes nonterminaler med attributter oversættes til Prolog, ved nok ved brug af `listing` som ovenfor.

```
?- listing(noun_phrase)
```

```
noun_phrase(Number, _1, _2) :-
          determiner(Number, _1, _3),
          noun(Number, _3, _2).
```

Som andre Prolog-programmer kan grammatikkerne naturligvis også bruges »baglæns«. Dvs. når man kan analysere sætninger, kan man også generere dem; der genereres i alt 126 sætninger.

```
?- phrase( sentence, S)
S = [the, man, eats];
S = [the, man, loves]:
...
S = [the, man, loves, the, woman];
S = [the, man, loves, the, women];
S = [the, man, loves, the, apple];
S = [the, man, loves, the, apples];
S = [the, man, sings, the, man];
...
S = [the, apples, sing, the, women];
S = [the, apples, sing, the, apple];
S = [the, apples, sing, the, apples];
No (more) solutions
```

Dette giver inspiration til et lille tankeeksperiment. Man kunne udstyre grammatikreglerne med nok et argument, som angav en eller anden universel trærepræsentation af den analyserede sætning. Universel i betydningen, at den kan bruges for ethvert af denne verdens sprog.[17] Skriv så yderligere en grammatik for dansk. Vi ville da forvente følgende forespørgsel med samt svar.

```
?- phrase( sentence(UniverseltTræ), [the, man, eats]),
   phrase( sætning(UniverseltTræ), S)


UniverseltTræ = ...
S = [manden, spiser]
```

Opgave 4.12 Definite klausulgrammatikker kan implementeres ved en oversættelse fra grammatikreglerne, til de tilsvarende Prologregler, som vi har eksemplificeret ovenfor. Skriv et prædikat, `oversæt(`*DGC-regel*`,` *Prolog-regel*, som foretager denne oversættelse.

[17] Der er et stærkt omdiskuteret spørgsmål blandt lingvister, hvorvidt det er meningsfyldt at forestille sig et sådan universelt sprog, som således skulle være uafhængigt af kultur og værdinormer o.m.a.

4.7.2 DCG som metasprog for programmeringssprog

De principper for at bygge fortolkere og oversættere i Prolog, vi har set tidligere, kan også anvendes ved DCG. På denne måde kan vi opnå en kobling af syntaksgenkendelse med den semantiske behandling. Vi skal dog være klar over følgende begrænsninger:

- DCG'er går i uendelig løkke, hvis man benytter venstrerekursive grammatikker, og for at undgå dette fænomen kan det være nødvendigt at skrive grammatikkens regler om på ikke særlig smuk måde.
- DCG'en skelner ikke mellem abstrakt og konkret syntaks, så reglerne må nødvendigvis afspejle en strukturel (dvs. konkret) grammatik. Kombineret med pinden overover, kan dette gå ud over læsbarheden.
- DCG'er har ikke en kobling til analyse af leksikalsk syntaks, programmer i objekstsproget skal gives som Prolog-lister.

Vi viser her en version som DCG af oversætteren fra aritmetiske udtryk til Mini-stak-og-variabel-maskinen, hvor vi har udtrykt den korrekte associativitet og præcedens i reglerne samtidigt med, at vi har undgået venstre-rekursion

Vi benytter er speciel syntaks, som gør det muligt i krøllede paranteser at indføje Prologkode i grammatikreglerne således, at man kan benytte hele Prologs regnekraft til at forme nonterminalernes attributter.

```
udtryk(K) -->
    sekundært_udtryk(K1),
    evt_mere_udtryk(K2),
    {K <- K1 + K2}.


evt_mere_udtryk(K) -->
    [+],
    sekundært_udtryk(K1),
    evt_mere_udtryk(K2),
    {K <- K1 + udregn(+) + K2}.


evt_mere_udtryk([]) --> [].


sekundært_udtryk(K) -->
    tertiært_udtryk(K1),
    evt_mere_sekundært_udtryk(K2),
    {K <- K1 + K2}.
```

```
evt_mere_sekundært_udtryk(K) -->
    [*],
    tertiært_udtryk(K1),
    evt_mere_sekundært_udtryk(K2),
    {K <- K1 + udregn(*) + K2}.


evt_mere_sekundært_udtryk([]) --> [].


tertiært_udtryk(K) -->
    [Tal],
    {integer(Tal), K <- stak(Tal)}.
```

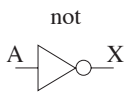
Som det fremgår, går noget af elegancen fløjten på grund af de underlige grammatikregler. Man kan opnå en bedre fremstilling ved først at benytte en DCG til at konstruere abstrakte syntakstræer, og derefter skrive en oversætter i Prolog, der tager udgangspunkt i abstrakte syntakstræer, som vi har set det tidligere.

Opgave 4.13 Foretag en håndsimulering af overstående oversætter anvendt på udtrykket repræsenteret ved Prolog-listen `[1,+,2,*,3]`.

5 Logiske kredsløb

Logiske kredsløb, som f.eks. beskrevet hos (Tanenbaum 1999, kap. 3), kan repræsenteres på elegant måde i Prolog. Indplaceret i forhold til kapitel 3, er der tale om en oversættelse fra det grafiske sprog af logiske kredsløb til Prolog, og det generelle princip i denne oversættelse beskrives her ved eksempler.

En given komponent (eller kredsløb) kan defineres som et prædikat, hvis argumenter repræsenterer komponentens (eller kredsløbets) ind- og udgående forbindelser. En »not gate«, for eksempel, er defineret ved følgende sandhedstabel.

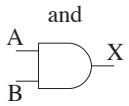


A	X
0	1
1	0

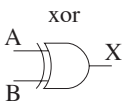
Det tilsvarende Prologprædikat `not(A, X)` kan defineres ved en række fakta, ét for hver række i tabellen.

```
not(nul, et ).
not( et, nul).
```

På tilsvarende måde kan vi definere »and gates« og »xor gates«, osv.



A	B	X
0	0	0
0	1	0
1	0	0
1	1	1

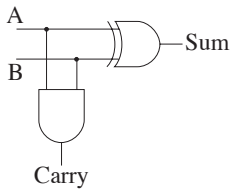


A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

De tilsvarende prædikater `and(A, B, X)` og `xor(A, B, X)` kan defineres som følger.

```
and(nul, nul, nul).          xor(nul, nul, nul).
and(nul,  et, nul).          xor(nul,  et,  et).
and( et, nul, nul).          xor( et, nul,  et).
and( et,  et,  et).          xor( et,  et, nul).
```

I og med de basale komponenter er repræsenteret som prædikater, kan vi bygge nye prædikater, som beskriver kredsløb. Betragt for eksempel følgende kredsløb for en såkaldt »half adder«.

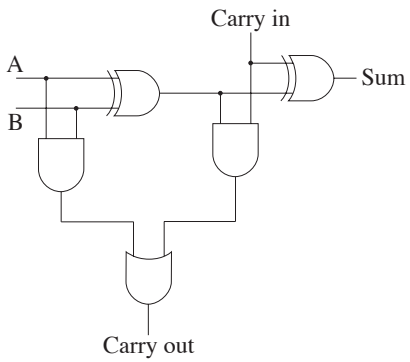


Den kan beskrives ved et prædikat, defineret som følger.

```
halfadder(A, B, Carry, Sum):-
    and(A, B, Carry), xor(A, B, Sum).
```

Prologs variable overtager her rollerne som ledninger. Som man måske kan erindre, er en elektrisk leder i elektricitetslæren defineret som et legeme, der har samme spænding overalt. Uanset om spændingen er påtrykt i den ene eller anden ende af lederen, så er spændingen den samme overalt.

Vi tager en mere kompliceret kreds, en »full adder«.



Her forekommer nogle rent interne ledninger, som ikke er forbundet med kredsens ind- og ud-ben, og derfor identificeres de med nogle nye, »lokale« variable, `X`, `Y` og `Z`.

```
fulladder(A, B, Carryin, Sum, Carryout):-
    xor(A, B, X),
    and(A, B, Y),
    and(X, Carryin, Z),
    xor(Carryin, X, Sum),
    or(Y, Z, Carryout).
```

Der er her tale om en model af et fysisk system, og vi kan bruge modellen til at forudsige egenskaber ved dette system ved at stille forespørgsler.

```
?- fulladder(nul, et, et, S, C).
S = nul,  C = et;
No (more) solutions
```

Modsat det fysiske system, kan vi også spørge om, hvilke mulige input, som kan resultere i givet output.

```
?- fulladder(X, Y, Z, nul, et).
X = nul,  Y = et,  Z = et;
X = et,  Y = nul,  Z = et;
X = et,  Y = et,  Z = nul;
No (more) solutions
```

Analogien mellem elektriske ledninger holder, sålænge komponenterne fungerer som rene matematiske funktioner, men i komponenter med hukommelse, f.eks. en »flip-flop« bryder analogien sammen.[1]

[1] Prøv evt. at beskrive en »flip-flop« (se Tanenbaum, 1990) efter principperne beskrevet her, og se, hvad der sker.

6 Sekventielle og imperative sprog

Vi udsætter her to grundlæggende paradigmer indenfor programmeringssprog for en nærmere undersøgelse. Når vi taler om sekventielle sprog, lægger vi specielt vægt på den sekventielle kontrol som er grundlæggende i von Neuman-arkitekturen og dermed i sædvanlige maskinsprog, men som vi også genfinder i sædvanlige imperative og objektorienterede sprog. Ved imperative sprog tænker vi på sprog med udtryk, assignment og kontrolstrukturer á la while, og som sådan også danne basis for de fleste udbredte prorgammeringssprog. Vi beskriver her to udvalgte, minimalistiske repræsentanter for de to sprogparadigmer ved definerende fortolkere efter de principper, som blev introduceret i afsnit 4.4. Der vises og diskuteres også en oversætter fra det imperative niveau til det sekventielle.

6.1 En definerende fortolker for et sekventielt sprog

Sproget for vor Mini-stak-og-variable-maskine benytter et kontrolmønster, som svarer til typiske von-Neuman-maskiner og deres efterfølgere. Et program er en sekvens af instruktioner, som udføres i den rækkefølge de optræder i programmet, med mindre en hopinstruktion udsiger, at den normale udførelse brydes og genoptages et andet sted i sekvensen bestemt ved en etikette. Udover hopinstruktioner, som varetager kontrollen, findes så de instruktioner, der, om man så må sige, udfører det egentlige arbejde. For Mini-stak-og-variable-maskinen har vi et udvalg at instruktioner, som foretager ændringer på en stak og et lager, som her er forenklet til at være en afbildning fra variable til talværdier. Vi får således ikke afspejlet den egenskab ved von-Neuman-maskinen, at programmet, der bliver udført, er placeret i samme lager som stakke og andre datastrukturer.

Vi udvikler her en definerende fortolker for Mini-stak-og-variable-maskinens sprog formuleret i Prolog. Når vi siger *definerende* fortolker, er det fordi, den giver en præcis definition af sprogets semantik og forhåbentlig skærper forståelsen heraf — og det, at den er en fortolker skrevet i et implementeret og kørende sprog, gør det nemt at eksperimentere med ændringer af sproget, f.eks. med nye kontrolinstruktioner.

Den samlede effekt af et program, resultatet af at have udført det, kan aflæses i den afsluttende stak og den afsluttende lager. Derfor er det naturligt at beskrive semantikken ved et prædikat `prog`, som beskriver en relation mellem programmer og stak+lager. Vi repræsenterer programmer ved lister af instruktioner som vist i det følgende, stakken repræsenteres som en liste af værdier med staktoppen mod venstre (hvilket passer fint sammen med Prologs listenotation) og lageret som en liste af strukturer af formen *variabel=værdi*. Vi vil forvente, et dette prædikat, når vi engang fået det defineret, vil udvise en opførsel svarende til disse eksempler.

```
?- prog([stak(3),stak(2),udregn(-)],S,L).
L = [],
S = [1]


?- prog([stak(4),gem(x)],S,L).
L = [x=4],
S = []
```

Operationer på stakken udtrykkes snildt ved hjælp af Prologs listenotation og for at udtrykke operationer på lageret, indfører vi to hjælpeprædikater. Prædikatet `hent` vil, givet en variabel og et lager, returnere variablens værdi i henhold til dette lager; hvis variaben ikke er registreret i lageret, returneres værdien `0`. Prædikatet `erstat` vil, givet variabel, værdi, og et lager, producere et nyt lager, hvor variablen er bundet til værdien (med en evt. gammel værdi slettet), og alle andre variable har bevaret deres værdi. Disse to prædikater defineres nemt ved følgende etuder udi listegymnastik.

```
hent(Var,X,L):-
    member((Var=X),L) -> true
    ;
    X = 0.


erstat(V,X,[],[(V=X)]).


erstat(V,X,[(V=_)|L],[(V=X)|L]):- !.
```

```
erstat(V,X,[Bind|L0],[Bind|L1]):-
    erstat(V,X,L0,L1).
```

Lad os for et øjeblik se bort fra hopinstruktioner og analysere, hvordan instruktioner iøvrigt udføres. Instruktionerne beskriver hver i sær simple ændringer på stak+lager, så vi kan modellere en enkelt instruktion, som en transformation fra et sæt stak+lager til et ny. En `hent(x)`-instruktion vil, givet en stak `S` og et lager `L`, producere ny stak `[Værdi|S]`, hvor der gælder `hent(x,Værdi,L)`; lageret er uændret, så vi kan give `[Værdi|S]` og `L` videre til næste instruktion i sekvensen.

Til beskrivelse af udførelse af sekvenser af instruktioner, herunder hele programmer, definerer vi et prædikat `sekvens`, som udtrykker såvel transformation af stak+lager som det, at give disse videre til næste instruktion. Dette prædikat har som argumenter en sekvens af instruktioner, et sæt af stak+lager i hvilket udførelsen af sekvensen startes, samt det resulterende stak+lager, som fremkommer ved at udføre hele sekvensen. Hver instruktion kan da beskrives ved en regel for `sekvens`-prædikatet, som udfører de relevante transformationer og giver »kontrollen« videre til de efterfølgende instruktioner ved et rekursivt kald; `hent`-instruktionen giver således anledning til følgende regel.

```
sekvens([hent(Var)|Rest], S0, L0, S1, L1):-
    hent(Var,X,L0),
    sekvens(Rest, [X|S0], L0, S1, L1).
```

Her står `S0`+`L0` for tilstanden umiddelbart før `hent`-instruktionen udføres, `[X|S0]`+`L0` for tilstanden umiddelbart efter, og `S1`+`L1` for den afsluttende tilstand, som resulterer af at udføre hele sekvensen angivet ved `[hent(Var) | Rest]`.

På denne måde kan vi beskrive alle instruktioner i sproget, men ikke hop-instruktionerne. Lad os betragte instruktion `hop(7)`, og antage, den står forrest i en den delsekvens, vi er nået til, dvs. vi står foran at skulle fortolke `[hop(7) |` *rest*`]`. Denne instruktion ændrer ikke ved tilstanden, men starter udførelsen et andet sted, nemlig ved den del af programmet som starter umiddelbart efter etiketten `7`, hvor den end måtte optræde i programmet. Men denne etikette kan optræde et vilkårligt sted i den samlede program og ikke nødvendigvis være at finde i `rest`. For at kunne beskrive hop-instruktioner er vi altså nødt til at slæbe hele programmet med, sådan at man kan finde ud af, hvad de benyttede etiketter faktisk refererer til.

Vi udvider derfor `sekvens`-prædikatet med nok et argument, som ikke ændrer sig undervejs, men blot fra starten af forsynes med hele programmet. Reglen for `hop` kommer nu til at se således ud, idet vi benytter `append`-prædikatet til at lokalisere sekvensen (kaldet `Fortsættelse`) efter etiketten.

```
sekvens([hop(E)|_], P, S0, L0, S1 ,L1):-
    append(_, [E|Fortsættelse], P),
    sekvens(Fortsættelse, P, S0, L0, S1, L1).
```

Denne regel giver således en kompakt og præcis beskrivelse af, hvad det vil sige at udføre en `hop`-instruktion i konteksten af et omgivende program (refereret til ved `P`).

Det betingede hop `n_hop` aflæser og fjerner et værdi på stakken, og afhængigt af denne vælges den normale sekvens, eller man fortsætter som ved `hop`. Dette indfanges af følgende to regler:

```
sekvens([n_hop(E)|_], P, [nej|S0], L0, S1 ,L1):-
    append(_, [E|Fortsættelse], P),
    sekvens(Fortsættelse, P, S0, L0, S1, L1).


sekvens([n_hop(_)|Rest], P, [ja|S0], L0, S1 ,L1):-
    sekvens(Rest, P, S0, L0, S1 ,L1).
```

En ting, man sandsynligvis ville have glemt at specificere, hvis ikke denne beskrivelse af et sprogs semantik samtidigt var en kørende fortolker, er, hvordan en etikette, når vi møder den i en sekvens af instruktioner, skal forstås. Svaret er enkelt: Ingen verdens ting, hvilket præciseres ved følgende regel, som nemt og elegant skipper henover etiketten.

```
sekvens([Etikette|Rest], P, S0, L0, S1 ,L1):-
    integer(Etikette),
    sekvens(Rest, P, S0, L0, S1 ,L1).
```

En anden triviel ting, der er så indlysende, at man kunne glemme den, hvis ikke man opdagede den ved første testkørsel, er at afslutte struktureret, når den sidste instruktion er passeret. Det klarer vi således:

```
sekvens([],_,S,L,S,L).
```

De øvrige instruktioner kan nu beskrives på ligefrem måde, idet vi skal huske at slæbe programmet uændret (og ubeset) med som argument mellem de enkelte rekursive kald.

```
sekvens([stak(N)|Rest], Prog, S0, L0, S1, L1):-
    sekvens(Rest, Prog, [N|S0], L0, S1, L1).


sekvens([hent(Var)|Rest], Prog, S0, L0, S1, L1):-
    hent(Var,X,L0),
    sekvens(Rest, Prog, [X|S0], L0, S1, L1).
```

```
sekvens([gem(Var)|Rest], Prog, [X|S0], L0, S1, L1):-
    erstat(Var,X,L0,Lx),
    sekvens(Rest, Prog, S0, Lx, S1, L1).

sekvens([udregn(+)|Rest], Prog, [X,Y|S0], L0, S1, L1):-
    YplusX is Y + X,
    sekvens(Rest, Prog, [YplusX|S0], L0, S1, L1).

sekvens([udregn(-)|Rest], Prog, [X,Y|S0], L0, S1, L1):-
    YminusX is Y - X,
    sekvens(Rest, Prog, [YminusX|S0], L0, S1, L1).

...

sekvens([udregn(>)|Rest], Prog, [X,Y|S0], L0, S1, L1):-
    (Y > X -> YstX = ja ; YsX = nej),
    sekvens(Rest, Prog, [YstX|S0], L0, S1, L1).

...
```

For at have specificeret et definerende fortolker for mangler vi blot at koble dette prædikat, `sekvens` med seks argumenter, til `prog`-prædikatet, som afbilder et program over i afsluttende stak+lager. Det gøres med følgende regel, som sætter de rigtige argumenter op, dvs. sørger for, at udførelsen starter ved første instruktion og med den rette starttilstand.

```
prog(Prog, StakSlut, LagerSlut):-
    sekvens(Prog,Prog,[],[], StakSlut, LagerSlut).
```

Vi har nu opnået at gøre Mini-stak-og-variabel-maskinen fra at være rent abstrakt til også at være en virtuel maskine, som det fremgår af følgende testkørsel.

```
?- prog( [   stak(23),
             gem(t),
          7, hent(x),
             stak(7),
             udregn(+),
             gem(x),
             hent(t),
             stak(1),
             udregn(-),
             gem(t),
             hent(t),
             stak(0),
             udregn(=),
             n_hop(7)], S, L).

L = [t=0,x=161],
S = []
```

Lad os opsummere, hvad vi ellers har opnået ved denne øvelse:

- Vi har formuleret et præcis model, som giver os en *teoretisk* forståelse af, hvad kontrol i maskinsprog er. Hvad etiketter er, hvad det vil sige at hoppe til dem, hvad det vil sige at udføre instruktioner i sekvens, hvordan stakinstruktioner fungerer osv.

- Vi har vist en *definerende* fortolker, hvilket vi vover at kalde, den fordi den 1) er formuleret i et sprog, som har en deklarativ semantik, der kan forstås uden at inddrage en underliggende implementation, og 2) den fremstår i en relativt læsbar og klar form, således at det er muligt at benytte den til at ræsonnere om den her definerede, abstrakte maskine.

Havde der istedet for et prototypisk maskinsprog været tale om et sprog til en ny og fancy brugerflade, kunne den definerende fortolker være brugt som et værktøj i designet af denne brugerflade. Vi kan eksperimentere med forskellige udformninger af instruktioner (eller hvad tingene nu det måtte hedde i sproget under betragtning), elaborere og præcisere deres semantik under stadig eksperimenteren, og vi har mulighed for løbende at undersøge, om sproget nu er tilstrækkeligt til at udtrykke de ting, vi gerne vil.

Opgave 6.1 I reglen for `udregn(-)`, udpeges de to øverste elementer af stakken ved at unifie denne med `[X,Y|S0]`, hvorefter man udfører subtraktionen `YminusX is Y - X`, hvilket kunne se lidt bagvendt ud ved første øjekast. Hvad ville det ændre ved maskinens overordnede opførsel, hvis vi i stedet udregnede `X - Y` — og analogt for alle andre operationer, der vedrører stakken?

Opgave 6.2 En oplagt mangel ved Mini-stak-og-variabel-maskinen er manglen på subrutiner, også kalde underprogrammer. Udvid fortolkeren med (og præcisér derved semantikken for) følgende instruktioner:

- `hop_sub(`*hop-etikette*`,`*retur-etikette*`)` — *retur-etikette* lægges som en værdi på stakken, hvorefter udførelsen genoptages ved *hop-etikette*.
- `retur` — det øverste element på stakken opfattes som en etikette, som der hoppes til.

Instruktionen `hop_sub` kan bruges til at implementere parameterløse procedurer, hvor *hop-etikette* svarer til procedurens start, og hvor *retur-etikette* angiver instruktionen umiddelbart efter `hop_sub`-instruktionen. Konstruér et eksempel på en sådan procedure og aftest den.

Opgave 6.3 (Fortsættelse af foregående opgave). Det ville være mere interessant, hvis subrutinerne kunne have parametre, så kunne man benytte dem til at implementere »pseudo-instruktioner« svarende til regneoperationer, som maskinen ikke har, men som implementeres ved subrutiner. Giv et forslag til, hvordan instruktioner til dette formål kunne se ud, og tilføj dem evt. til fortolkeren og aftest på et eksempel.

6.2 En definerende fortolker for while-programmer

Kontrolstrukturer som `if` og `while`-strukturer kan godt forstås som syntaktisk sukker lagt over på et maskinsprog, og tilsvarende for assignment og udtryk. Eksempelvis kan tænke på

```
while x<7 do x:= x+1
```

som en elegantere måde at skrive den mere udførligere

```
10, hent(x), stak(7), udregn(<), n_hop(11),
    hent(x), stak(1), udregn(plus),
    gem(x),
11
```

Der er set eksempler på lærebøger (specielt ældre af slagsen), som forklarer sin læser om while-løkker på denne måde. Men vi kunne stille os selv det spørgsmål, om ikke et sprog med `while`-løkker m.v. ikke kan opfattes som et sprog med sit eget velværd, som kan forklares på nogle andre og mere abstrakte præmisser end ved hop-instruktioner og den slags. Svaret er naturligvis

»jo«, og det kan vi begrunde ved at give en sådan abstrakt specifikation, enten ved at henvise til en aksiomatisk beskrivelse vha. Hoare-logik eller, som her, vha. en definerende fortolker, som benytter sig af rekursion.

Lad os først give en abstrakt syntaks for det sprog, vi vil omtale som while-programmer. Den kan vi opskrive på forskellig måde, f.eks. skrive et Prologprogram, som netop accepterer (de termer, vi vil forstå som) abstrakte syntakstræer for dette sprog. Vi benytter her en abstrakt BNF, som angiver en repræsentation af sprogets abstrakte syntakstræer som Prologtermer:

⟨program⟩::= ⟨sætning⟩

⟨sætning⟩::= `tom`
| ⟨sætning⟩ `;` ⟨sætning⟩
| ⟨variabel⟩ `:=` ⟨udtryk⟩
| `if(`⟨betingelse⟩`,` ⟨sætning⟩`,` ⟨sætning⟩`)`
| `while(`⟨betingelse⟩`,`⟨sætning⟩`)`

⟨udtryk⟩ ::= ... ⟨heltal⟩ *med* `+`, `-`, `*` *og* ⟨variable⟩

⟨betingelse⟩ ::= `sand` | `falsk`
| ⟨udtryk⟩ = ⟨udtryk⟩ | *...og andre sammenligninger*
| ⟨betingelse⟩ `/\` ⟨betingelse⟩
| ⟨betingelse⟩ `\/` ⟨betingelse⟩
| `\+` ⟨betingelse⟩

⟨variabel⟩ ::= `a` | `b` | ··· | `å`

Vi benytter symbolet `\+` for negation af betingelser i analogi med Prologs syntaks.

Når vi benytter de operatordefinitioner, som er standard i Prolog kan vi skrive (en Prologterm, som repræsenterer et abstrakt syntakstræ for) et program, der ved hjælp af Euklids algoritme udregner største fælles divisor for 221 og 493 på følgende måde:

```
a:= 221 ; b:= 493 ;
while( a =\= b,
       if( a > b,
           a:= a-b,
           b:= b-a))
```

For at kunne beskrive semantikken antager vi samme slags lager som vi benyttede for fortolkeren for Mini-variabel-og-stak-maskinen i det foregående afsnit, dvs. afbildninger fra variable til værdier. While-sproget er et prototypisk og minimalistisk eksempel på, hvordan et *imperativt sprog* er opbygget: Det indeholder operationer, som foretager simple tilstandsændringer (her: »`:=`«), som ændrer én variabels værdi ad gangen), og disse styres ved konstruktioner, som udtrykker gentagelser og betinget udførelse.

Vi vil nu diskutere den overordnede udformningen af de semantiske relationer for de enkelte syntaktiske kategorier. Udtryk indeholder variable, så deres værdier afhænger af det aktuelle lager, og denne observation for os til at foreslå en semantisk relation af formen $\langle$*lager*, *heltal*$\rangle$. Tilsvarende for betingelser $\langle$*lager*, {`sand`, `falsk`}$\rangle$. Sætninger beskriver ændringer på lageret, og involvere tilsyneladende ikke andre ting, så her foreslår vi $\langle$*lager*, *lager*$\rangle$ (svarende til lager-før og lager-efter). For hele programmer antager vi, at der altid sættes en fast tilstand op (hvor alle variable har værdien 0), og den semantiske relation for given program er dets sluttilstand. Vi får da følgende semantiske prædikater:

`program(`*program*`,` *lager*`)`
`sætning(`*sætning*`,` *lager*`,` *lager*`)`
`udtryk(`*udtryk*`,` *lager*`,` *heltal*`)`
`betingelse(`*betingelse*`,` *lager*`,` {`sand`, `falsk`}`)`

Vi præsenterer nu et sæt semantiske regler for while-sproget. Programmer (dvs. den sætning, som et program består af), startes i et tomt lager (hvor så `hent`-prædikatet præsterer en illusion af, at enhver variabel har værdien `0` fra starten):

```
program(P, Lager) :- sætning(P, [], Lager).
```

Vi bemærker, at denne regel passer perfekt på definitionen af, hvad en abstrakt maskine er. Prædikatet `program` fungerer som en sådan, det »indlæser« et program og bestemmer den afsluttende hukommelse.

Den tomme sætning, har en veldefineret betydning: Den passerer tilstanden videre uændret:

```
saetning(tom, L, L).
```

Værditilskrivning eller assignment består i at udregne et udtryks værdi og opdatere variablen:

```
sætning( (Var := Udtryk), L1, L2):-
    udtryk( Udtryk, L1, Værdi),
    erstat( Var,Værdi,L1,L2).
```

Sekvens af sætninger noteret ved semikolon plejer vi at tænke på som »gør først dit og dernæst dat«. Intuitionen af »først« og »dernæst« kan defineres præcist på følgende måde:

```
sætning( (S1 ; S2), L1, L3):-
    sætning(S1, L1, L2),
    sætning(S2, L2, L3).
```

Den betingede `if`-sætning beskrives således:

```
sætning( if( Bet, Sætning1, Sætning2), L1, L2):-
    betingelse(Bet, L1, Værdi),
    (Værdi = sand ->
         sætning(Sætning1, L1, L2)
     ;
         saetning(Sætning2, L1, L2)).
```

Den semantiske relation for en `if`-sætning beskriver altså en udregning af betingelsen i dens før-tilstand, og afhængigt af resultatet bestemmes dens eftertilstand af den ene eller den anden af de to delsætninger.

Reglen for `while`-sætningen ligner den for `if`-sætning, men indeholder et rekursivt kald:

```
sætning( while( Bet, Sætning), L1, L2):-
    betingelse(Bet, L1, Værdi),
    (Værdi = sand ->
       sætning((Sætning ; while( Bet, Sætning)), L1, L2)
     ;
       L1 = L2).
```

Hvis betingelsen er falsk, er start-tilstand=slut-tilstand, ellers udføres kroppen én gang, og dernæst hele `while`-sætningen igen. Det kunne måske umiddelbart se ud som en uendelig løkke, når man forsøger at løse problemet bestående i at udføre `while(Bet, Sætning)` ved et tilsyneladende større delproblem, nemlig at udføre `Sætning ; while( Bet, Sætning)`. Men her skal man huske på, at det (forhåbentligt) ikke er det samme lager, som dukker up, når betingelsen beregnes for anden gang. Men omvendt, for de `while`-løkker, der er programmeret så dumt, at de vil gå i uendelig løkke, må den

definerende fortolker vel også skulle gøre det for at være korrekt.[1] Prøv at håndsimulere nogle simple `while`-løkker vha. denne fortolker.

Semantiske relationer for udtryk defineres, som vi har set tidligere i afsnit 4.4.2, blot skal vi i det imperative sprog slæbe lageret med rundt til alle underudtryk, således at variable kan finde deres værdier.

```
udtryk( Variabel, L, V):- atom(Variabel),
    hent(Variabel,V,L).
```

De udtryk og betingelser, som er konstanter, repræsenterer sig selv uanset hvad lageret måtte mene — mere præcist, til hver syntaktisk konstant (i `while`-sprogets abstrakte syntaks) hører en entydig konstant i det semantiske metasprog (dvs. her: Prolog).

```
udtryk( Tal, _, Tal):- integer(Tal).
betingelse( sand, _, sand).
betingelse( falsk, _, falsk).
```

Diverse operatorer opfører sig, som vi forventer:

```
udtryk( (Udtryk1 + Udtryk2), L, Res):-
    udtryk( Udtryk1, L, V1),
    udtryk( Udtryk2, L, V2),
    Res is V1 + V2.


...
betingelse( (Udtryk1 = Udtryk2), L, Res):-
    udtryk( Udtryk1, L, V1),
    udtryk( Udtryk2, L, V2),
    (V1 = V2 -> Res = sand
      ;
      Res = falsk).


...
betingelse( (Betingelse1 /\ Betingelse2), L, Res):-
    betingelse( Betingelse1, L, V1),
    betingelse( Betingelse2, L, V2),
    (V1 = falsk -> Res = falsk
     ;
     V2 = falsk -> Res = falsk
     ;
     Res = sand).
```

[1] Hmmmm, hvad mon der menes her? Giver det mening at tale om, at en definerende fortolker (eller en hvilken som helst anden definition) er korrekt eller ej? Overvej!

```
betingelse( (\+ Betingelse), L, Res):-
    betingelse( Betingelse, L, V),
    (V = sand -> Res = falsk
     ;
     Res = sand).
```

Summa summarum: Vi har nu opnået at en klar definition af et lille imperativt sprog med while-løkker. Vi kan bruge denne definition (hvis ellers vi er tilfreds med den), som et udgangspunkt til at vurdere spørgsmål af karakteren »Er denne og hin påståede implementation/beskrivelse af en while-løkke korrekt?«. Vi kunne f.eks. spekulere på, om den skitse vi gav i starten af dette afsnit, hvor vi karakteriserede betydningen af while-sætninger i termer af hop og etiketter faktisk er korrekt, eller om vi har fået kludder i etiketterne. Eller vi kan benytte den som udgangspunkt for at undersøge, om den oversætter for while-programmer, vi præsenterer i næste afsnit, faktisk producerer korrekte kodesekvenser, dvs. om den opfylder de korrekthedsbetingelser, vi indbyggede i definitionen af, hvad en oversætter er, i kapitel 3.

Mest interessant i tilfældet med while-løkker er vel, at vi har en simpel og veldefineret beskrivelse af den, som er både intuitivt overkommelig og matematisk præcis (fordi definitionen er skrevet i en del af Prolog med en velkendt logisk semantik). En anden pointe ved den definerende fortolker er også, at den kan fungere som et dejligt prototypeværktøj. OK, omkring while-løkken er der ikke så meget at rafle om, men ser vi f.eks. på diverse forekommende varianter af »for«-sætninger, så kan der være behov for eksperimenter inden man, i rollen som designer af et nyt programmeringssprog, lægger sig fast på en bestemt version.

Opgave 6.4 Udvid nu while-sproget med en ny type udtryk af formen

`result_is(` *sætning*`,` *variabel*`).`

Idéen er, at sætningen udføres, hvorefter variablens værdi angiver den samlede udtryks værdi. Typisk vil sætningen foretage assignments til den angivne variabel, og den kan måske også påvirke andre variable. Vi har altså indført en slags udtryk, som kan have sideeffekter. Hvordan påvirker det den semantiske relation for udtryk, hvordan kommer den tilsvarende semantiske regel til at se ud, og hvilke ændringer skal der i øvrigt foretages i fortolkeren? Foretag de nødvendige ændringer, og brug den definerende fortolker til at finde ud af, hvad resultatet af følgende program er:

```
x:= 7; x:= result_is( x:= x+1 , x)
```

Opgave 6.5 While-sproget mangler en fundamental konstruktion, som vi forbinder med imperative sprog, og der er brugerdefinerede funktioner og procedurer. Giv forslag til, hvordan den abstrakte syntaks kan udvides, så disse ting kommer med. Hvilke argumenter skal de semantiske prædikater udstyres med, for at betydningen af disse konstruktioner kan modelleres, og hvilke nye argumenter skal evt. tilføjes til de eksisterende prædikater? Vend evt. tilbage til denne opgave, når du har læst kapitel 7 og skriv en definerende fortolker for det således udvidede while-sprog.

6.3 Oversættelse af while-programmer

En oversætter, som vi definerede det, er en abstrakt maskine, hvis karakteristiske sprog svarer til kildesproget og hvis »semantiske funktion« udtrykker sprogets semantik indirekte ved en oversættelse til et anset sprog. Dette i modsætning en sædvanlig abstrakt maskine, f.eks. beskrevet ved en definerende fortolker, som direkte bestemmer den semantiske funktion $\equiv$ den semantiske relation.

Vi viser her en oversætter fra while-programmer til Mini-stak-og-variabel-maskinsprog. Denne oversætter illustrerer dog kun en meget lille del af det, man finder i »rigtige« compilere for realistiske sprog, bl.a. fordi der ikke er erklæringer eller typer i vore while-sprog, så vores oversætter ikke behøver at arbejde med symboltabeller. Den klassiske bog om compilere er (Aho, Sethi, Ullman, 1986) har et glimrende og lettilgængeligt oversigtskapitel (og rummer iøvrigt stort set alt, hvad der er værd at vide om compilere).

Oversætteren her kan vi formulere meget lig den fortolker vi så i det forrige afsnit. Her afbilder vi blot ikke syntakstræer over i semantiske relationer, men kodesekvenser, dvs. for hver syntaktisk kategori har vi et prædikate *kategori*`(`*abstrakt-syntakstræ*`,` *kode*`)`, og for at formulere oversættelsesreglerne benytter vi de hjælpeværktøjer, vi indførte i afsnit 4.3.3.

Reglerne for programmer, og de simple sætningstyper er lige ud ad landevejen:

```
program(P, K):- saetning(P, K).


saetning(tom, []).


saetning( (S1 ; S2), K):-
    saetning(S1, K1),
    saetning(S2, K2),
    K <- K1 + K2.
```

```
saetning( (Var := Udtryk), K):-
    udtryk( Udtryk, K1),
    K <- K1 + gem(Var).
```

If-sætninger oversættes, som vi har diskuteret på et tidligere tidspunkt, med en etikette til at markere slutningen og med et betinget hop til denne slutning, hvis betingelsen (når den engang altså måske bliver udført) er falsk.

While-sætningen er igen helt analog, nu med to etiketter, én til at angive starten af løkken (så der kan hoppes tilbage til den) og én til at angive slutningen.

```
saetning( while( Bet, Saetning), K):-
    betingelse(Bet, Kbet),
    saetning( Saetning, K1),
    ny_etikette(Estart), ny_etikette(Eslut),
    K <-  Estart + Kbet +
                   n_hop(Eslut) +
                   K1 +
                   hop(Estart) +
          Eslut.
```

Lad os nu gå videre med oversættelse af udtryk. I en fortolker kunne vi udtrykke betydningen af en konstant ved anden konstant, men i oversætteren bliver det lidt indirekte: Vi oversætter en konstant til en instruktion, som, hvis den engang bliver udført, vil lægge en konstant på stakken.

```
udtryk( Tal, K):-
    integer(Tal),
    K <- stak(Tal).


betingelse( sand,  stak(ja)).


betingelse( falsk, stak(nej)).
```

For de logiske konstanter er det tydeligt, at der foregår en oversættelse af konstanterne, da de nu tilfældigvis hedder noget forskelligt i de to sprog. Aritmetiske og logiske operatorer beskrives lige ud ad landevejen:

```
udtryk( (Udtryk1 + Udtryk2), K):-
    udtryk( Udtryk1, K1),
    udtryk( Udtryk2, K2),
    K <- K1 + K2 + udregn(+).


...
```

```
betingelse( (Udtryk1 = Udtryk2), K):-
    udtryk( Udtryk1, K1),
    udtryk( Udtryk2, K2),
    K <- K1 + K2 + udregn(=).


betingelse( (Betingelse1 /\ Betingelse2), K):-
    betingelse( Betingelse1, K1),
    betingelse( Betingelse2, K2),
    K <- K1 + K2 + udregn(/\).


...
betingelse( (\+ Betingelse), K):-
    betingelse( Betingelse, K1),
    ny_etikette(L), ny_etikette(Slut),
    K <- K1 + [     n_hop(L),
                    stak(nej),
                    hop(Slut),
                L, stak(ja),
                Slut
              ].
```

Den sidste regel, for negation i logiske udtryk, er lidt mere interessant end de andre. Her er vi i den situation, at Mini-stak-og-variabel-maskinsproget kun har binære operationer og således ikke har noget, der svarer til negation af en logisk værdi. Her må vi supplere med et længere stykke kode, som udfører beregningen (der foreslås en alternativ oversættelse i en opgave nedenfor).

Summa summarum: Vi har nu en kørende oversætter fra det »høj-niveau« imperative sprog af while-programmer til et sekventiel maskinsprog. Oversættelser produceret af traditionelle compilere fungerer meget lig det, vi har vist, omend compilere ofte lave diverse krumspring, herunder transformerer og ændrer oversat kode, i optimeringsøjemed. Men i tilfælde, hvor compileren ikke kan få øje på et interessant optimeringspotentiale, vil man kunne sammenligne de to stort set linie for linie. I det helt specielle tilfælde, hvor vi benytter et målsprog for oversættelsen, som er relativt simpel, kan man også opfatte oversætteren som en alternativ semantisk definition.

Til slut viser vi et eksempel på, hvordan oversætteren kan sættes sammen med en definere fortolker, således at vi summa summarum opnår en kørende implementation af sproget beskrevet gennem oversætteren. Vi viser her udregning af den største fælles divisor for `221` og `493` ved følgende forespørgsel til Prolog, hvor `program` refererer til oversætteren while-programmer $\rightarrow$ Mini-stak-og-variabel-maskinsprog og `prog` til fortolkeren for sidstnævnte.

```
?- program( ( a:= 221 ; b:= 493 ;
              while( a =\= b,
                     if( a > b,
                         a:= a-b,
                         b:= b-a))), K),
   prog( K, _, L).
```

Opgave 6.6 Tilføj en oversættelsesregel svarende til de `result_is`-udtryk, som blev indført i opgave 6.4 ovenfor.

Opgave 6.7 Udvid while-sproget med repeat-løkke, og udvid såvel fortolkeren som oversætteren til at håndtere dem. (Det tænkes på repeat-løkke svarende til, hvad der findes i Pascal. — Hvis du ikke lige kan huske, hvordan den ser ud, så slå den op i en bog eller spørg en ældre medborger).

Opgave 6.8 Følgende pricip kan bruges til en mere elegant oversættelse af negation på boolske udtryk:

```
K <- K1 + [stak(nej), udregn(=)]
```

Er den bedre end den, der blev brugt i oversætteren ovenfor — og hvad mon der egentligt menes med »bedre«?

Opgave 6.9 Man kunne forestille sig en anden repræsentation af logiske værdier: `falsk` repræsenteres ved tallet `0`, og et vilkårligt tal `>0` kan bruges til at repræsentere `sand`. Skitsér, hvordan oversættelsesreglerne for `betingelse`r så skal se ud.

Opgave 6.10 Kan man forestille sig en oversætter fra Mini-stak-og-variabel-maskinsprog til while-programmer? Skitsér, for nogle små Mini-stak-og-variabel-maskinprogrammer, hvordan de tilsvarende while-programmer kunne komme til at se ud. — De erfaringer, vi kan drage heraf synes ikke at have praktisk relevans, eller er der? Kan vi konkludere noget på det teoretisk plan.

7 Funktionsorienterede sprog, procedureabstraktion

Historisk set er Lisp et meget vigtigt sprog, som stammer fra slutningen af halvtredserne og begyndelsen af tresserne (McCarthy, 1960, McCarthy et al, 1962). I sin grundstruktur er det et funktionsorienteret sprog baseret på Churchs lambda-kalkyle (Church, 1941). Lambda-kalkylen har herigennem og på mange andre måder haft stor betydning for datalogien. Samtidigt med at Lisp har en simpel og matematisk veldefineret kerne, så er »rigtig Lisp« et af de sprog, som tillader de stærkeste sideeffekter, så man f.eks. kan komme til at modificere de basale funktioner i systemet. Om dette egentligt var McCarthys intention, eller det måske snarere afspejler, hvor meget »man« vidste om programmeringssprog på det tidspunkt, er et ofte diskuteret emne. Lisp var et af de første sprog, hvor de fundamentale dataobjekter var symboler — og ikke tal — og på denne måde kan man se Lisp en forløber for Prolog. Til de, som vil vide mere om Lisp, henvises til (Abelson, Sussman, 1985) og (Winston, Horn, 1989). I det følgende studerer vi et lille Lispagtigt sprog, som vi kalder MiniLisp. Dette kapitel giver dog ikke en dækkende introduktion til funktionsprogrammering, da det er kommet meget til siden, f.eks. dovne beregninger (eng.: lazy evaluation) og avancerede typebegreber. Læsere med en interesse for funktionsprogrammering kunne få nytte af at studere det væsentligt nyere funktionsprogrammerinssprog ML.

Vi får illustreret begrebet abstraktionsmekanismer (her funktionsdefinitioner), og hvordan disse implementeres, og vi får anledning til at diskutere forskellige parameteroverførselsmekanismer. Vi beskriver MiniLisp ved en definerende fortolker og benytter yderligere denne til at illustrere begrebet transformation, som er af stor betydning optimerende compiler.

7.1 Syntaks og semantik af programmeringssproget MiniLisp

MiniLisp er et sprog af udtryk, dvs. enhver frase i sproget er et udtryk. For nemheds skyld, benytter vi begreber fra Prolog til at definere syntaksen af disse udtryk.

Definition. Et MiniLisp-udtryk er et atom (incl. tal, m.v.) eller en liste af nul eller flere elementer, som hver især er MiniLisp-udtryk.

Semantikken af MiniLisp er således, at et udtryk kan have en værdi. Denne værdi er igen et udtryk. Det er dog kun nogle udtryk, som har en værdi, andre ikke.

Eksempel.

> 7 forventes at have værdien 7.
> [plus,1,2] forventes at have værdien 3.
> [[a,b],[c,d]] har ikke nogen værdi.

Værdien af et udtryk afhænger af den aktuelle hukommelse, som rummer variabelbindinger og funktionsdefinitioner. Udover at give anledning til en værdi, kan et udtryk også modificere hukommelsen.

For udtryk, som er atomer, gælder følgende.

tal
: Værdien af et *tal* er *tal*let selv.

[], t
: Værdien af disse specielle atomer er atomet selv. (NB: [] hedder almindeligvis »nil« i Lisp)

øvrige atomer
: Når der spørges på værdien af et atom forskelligt fra de ovennævnte, opfattes dette som en variabel. Dvs. den senest bundne værdi for dette atom returneres. Det opfattes som en fejl, hvis atomet endnu ikke er bundet til en værdi.

For listeudtryk gælder generelt, at første element er en angivelse af, hvordan værdien beregnes. Der gælder følgende.

[quote, X]
: Værdien af dette udtryk er ganske enkelt X (dvs. evalueringen undertrykkes).

[plus, X, Y]
: Værdien fås ved først at finde værdien for X, derefter værdien for Y, og lægge disse sammen.

[minus, X, Y], [times, X, Y]
— analogt!

[equal, X, Y]
Hvis værdierne af X og af Y er ens, så er værdien af dette udtryk t, ellers [].

[car, X]
Såfremt værdien af X er en liste, er værdien af »car«-udtrykket hovedet af denne liste.

[cdr, X]
Såfremt værdien af X er en liste, er værdien af »cdr«-udtrykket halen af denne liste.

[cons, X, Y]
Værdien af »cons«-udtrykket er en liste, hvis hovede er værdien af X, og hvis hale er værdien af Y.

[setq, *atom*, X]
atom (opfattet som variabel) bindes til værdien af X. Værdien af setq-udtrykket er værdien af X.

[progn, $X_1, \ldots, X_n$] (n$\geq$1)
Udtrykkene $X_1, \ldots, X_n$ udregnes[1] fra venstre mod højre; værdien af hele udtrykket er værdien af X_n.

[if, X, Y, Z]
Hvis værdien af X er forskellig fra [], fås værdien af »if«-udtrykket som værdien af Y, ellers som værdien af Z.

[*atom*, X]
Såfremt der findes en brugerdefineret funktion med navn svarende til *atom*, så anvendes denne funktion på værdien af X.

[defun, *navn*, *parameter*, *krop*]
navn og *parameter* skal være atomer, *krop* et udtryk. Der oprettes en funktion, som kaldes på følgende måde,

[*navn, X*],

og hvis værdi fås som følger. Parametren[2] (dvs. atomet *parameter* opfattet som variabel) bindes temporært til værdien af X,værdien af funktionskaldet fås som værdien af *krop* (i den aktuelle hukommelse!), og derefter slettes ovennævnte variabelbinding.

Ovenstående kan altså læses som en definition af MiniLisps syntaks og semantik. Semantikken bestemmer for nogle udtryk en værdi (f.eks. for [plus,

[1] Pragmatisk bemærket, for deres sideeffekters skyld.

[2] Traditionelt betegnes atomet *parameter* den formelle parameter og en faktisk forekommende værdi af X betegnes en aktuel parameter.

1, 2]), hvorimod andre udtryk (f.eks. [[a,b], [c,d]]) ikke har defineret en værdi. Værdien af et udtryk som [fak, x] afhænger af den aktuelle hukommelse.

Hvis man går beskrivelsen af [defun, . . .] igennem, vil man også kunne se, at der ikke er klart defineret, hvordan referencer til variable inde fra kroppen skal behandles. — Var der tale om et sprog, som skulle markedsføres, skulle dette selvfølgelig gøres præcist.

7.2 En fortolker for MiniLisp skrevet i Prolog

Fortolkeren er defineret som et prædikat, kaldet `lisp`, med to argumenter, hvor det første forventes givet som et udtryk, og det andet som en variabel i hvilken udtrykkets værdi returneres.

Bemærk, at reglerne for »setq« og »defun« benytter sig af Prologs indbyggede `asserta`- og `retract`-prædikater. `asserta` tilføjer en klausul til databases (dvs. til programmet!), det afsluttende »`a`« i prædikatnavnet betyder, at klausulen placeres foran alle andre; der findes også en `assertz`, som sætter klausulen sidst. `retract` sletter klausuler.

Det vil altså sige, at vi benytter Prologs hukommelse/database/program til at rumme MiniLisps hukommelse. Det medfører bl.a., at en funktionsdefinition oprettet i et kald af `lisp` kan benyttes i efterfølgende kald. Man kunne selvfølgelig have programmeret hukommelsen som et ekstra argument, som slæbtes med i alle rekursive kald af prædikatet `lisp`. Dette argument måtte så konsulteres, når man havde brug for en variabel eller en brugerdefineret funktion. Man kan måske sige, at den valgte løsning understreger Lisps karakter af at være et meget dynamisk sprog.

Kildeteksten i uredigeret form

```
%% lisp( <lisp-udtryk>, <værdi> )


lisp( [quote, X], X).


lisp( [plus, X, Y], Værdi):-
    lisp(X, Xværdi),
    lisp(Y, Yværdi),
    Værdi is Xværdi + Yværdi.
```

```
lisp( [minus, X, Y], Værdi):-
    lisp(X, Xværdi),
    lisp(Y, Yværdi),
    Værdi is Xværdi - Yværdi.

lisp( [times, X, Y], Værdi):-
    lisp(X, Xværdi),
    lisp(Y, Yværdi),
    Værdi is Xværdi * Yværdi.

lisp( Tal, Tal):-
    integer(Tal).

lisp([], []).

lisp(t, t).

lisp( [equal, X, Y], Værdi):-
    lisp(X, Xværdi),
    lisp(Y, Yværdi),
    (Xværdi = Yværdi -> Værdi = t; Værdi = []).

lisp( [car, X], Værdi):-
    lisp(X, Xværdi),
    (Xværdi = [Værdi | _] -> true
     ;
       nl, write('Cdr of non-list: '),
       write(Xværdi), abort).

lisp( [cdr, X], Værdi):-
    lisp(X, Xværdi),
    (Xværdi = [_ | Værdi] -> true
     ;
       nl, write('Cdr of non-list: '),
       write(Xværdi), abort).

lisp( [cons, X, Y], Værdi):-
    lisp(X, Xværdi),
    lisp(Y, Yværdi),
    Værdi = [Xværdi | Yværdi].
```

```
% To regler for progn:

lisp( [progn, X], Xværdi):-
   !, lisp(X, Xværdi).

lisp( [progn, X | MereX], Værdi):-
    lisp(X, _),
    lisp( [progn | MereX], Værdi).

lisp( [setq, V, X], Xværdi):-
    lisp(X, Xværdi),
    asserta( (
             lisp( V, Xværdi):- !
           )  ).

lisp( [if, B, X, Y], Værdi):-
    lisp(B, Bværdi),
    ( Bværdi = [] -> lisp(Y, Værdi)
      ;
      lisp(X, Værdi) ).

lisp( [defun, F, Param, Krop], F):-
    asserta( (
             lisp( [F, Arg], Værdi):-
               !,
               lisp( [setq, Param, Arg], ArgV),
               lisp( Krop, Værdi),
               retract( (lisp(Param, ArgV):- !) )
           )  ).
```

Klausulen for `defun` benytter Prologprimitivet `asserta`, som vi har diskuteret tidligere. Problemerne med sammenblanding af objektvariable (dvs. her variable, som skal »leve« i den `assert`ede klausul) og metavariable (som her står for stumper af Prologprogramtekst, som bliver skabt på udføresestidspunktet) er påtrængende. Specielt her, da vi kald af `asserta` (udløst af `lisp( [setq, ...], ...)`) og `retract` inde i kroppen af, som `assert`es på det yderste niveau. Lad os derfor forklare den i detaljer.

Den regel, som konstrueres med henblik på at blive `assert`et, er beskrevet følgende udtryk.

```
lisp( [F, Arg], Værdi):-
  !,
  lisp( [setq, Param, Arg], ArgV),
  lisp( Krop, Værdi),
  retract( (lisp(Param, ArgV):- ! )
```

Ved en omhyggelig granskning af den samlede fortolkers dynamiske egenskaber kan man se, at variablen `Arg` fungerer som et navn på en variabel i den klausul, som bygges. Variablen `F`, derimod, fungerer som en variabel i den omgivende klausul. `F` vil under udførelsen af denne blive bundet til et Prologatom, der således giver et bidrag til teksten for den klausul, som er under konstruktion. Endnu værre bliver det, når vi kigger på variablen `ArgV`, som indgår i kaldet af `retract`. Prologvariable bliver altså brugt til adskillige og forskellige formål, sådan som vi bruger Prolog. Mere præcist, de fungerer såvel som metavariable (`F`), som navne på objektvariable (`Arg`), uden at det fremgår eksplicit, hvad der er hvad.

Kørselseksempler

Et simpelt test med udregning af et aritmetisk udtryk.

```
?- lisp([plus,1,[times,3,4]], V)

V = 13
```

Lad os udføre en »setq«, som skulle lave sideeffekt på databasen (dvs. programmet!)

```
?-  lisp([setq, x, 7], K)

K = 7
```

Vi bruger det indbyggede »`listing`«-prædikat til at se, hvad vi har liggende.

```
?- listing(lisp)

lisp(x, 7) :-
  !.

lisp([quote, X], X).

...  Osv, alle de klausuler, vi selv havde skrevet.

yes
```

Dvs. den globale variabel `x` er beskrevet/implementeret på samme måde som de specielle, prædefinerede `t` og `[]`.

```
?-  lisp( x, K)

K = 7

?-  lisp([plus, x, x], K)

K = 14
```

Mere interessante sideeffekter ses ved definition af en ny funktion.

```
?- lisp([defun, kvadrat, n, [times,n,n]], K)

K = kvadrat
```

Lad os lige kontrollere . . .

```
?- listing(lisp)

lisp([kvadrat, _1], _2) :-
  !,
  lisp([setq, n, _1], _3),
  lisp([times, n, n], _2),
  retract((lisp(n, _3):-!)).

lisp(x, 7) :-
  !.          -- vor globale variabel fra før . . .

lisp([quote, X], X).
. . . Osv, alle de klausuler, vi selv havde skrevet.

yes
```

Dvs. denne brugerdefinerede funktion er beskrevet på samme måde som alle andre funktioner i den aktuelle Lispdialekt (!).

```
?- lisp([kvadrat, 6], K)

K = 36
```

Et eksempel på en rekursiv funktion, den velkendte fakultetsfunktion:

```
?- lisp([defun,fak,n,
          [if, [equal,n,0], 1,
             [times,n,[fak,[minus,n,1]]]]], F)


F = fak
```

Lad os lige kontrollere ...

```
?- listing(lisp)


lisp([fak, _1], _2) :-
  !,
  lisp([setq, n, _1], _3),
  lisp([if, [equal, n, 0], 1,
         [times, n, [fak, [minus, n, 1]]]], _2),
  retract((lisp(n, _3):-!)).


lisp([kvadrat, _1], _2) :-
  ... osv.
yes
```

Dvs. denne brugerdefinerede fakultetsfunktion er beskrevet på samme måde som alle andre funktioner i den aktuelle Lispdialekt (!).

```
?- lisp([fak, 3], K)


K = 6


?- lisp([fak, 40], K)


K = 8.159152832478977343e47
```

Opgave 7.1 Benyt beskrivelsen i afsnit 7.1 til at forudsige værdierne af nedestående MiniLisp-udtryk. Hold herunder rede på de enkelte deludtryks værdier og sideeffekter (dvs. ændringer i hukommelsen).

[times,5,[plus,2,3]]

[plus, [setq,x,3], x]

[plus, [progn, [defun, plusén, n, [plus,n,1]], 5],
    [plusén, 8]]

Opgave 7.2 Betragt fakultetsfunktionen fra kørselseksemplerne ovenfor. Opgaven går ud på at følge fortolkerens udførelsen af udtrykket [fak, 4], hvor man holder rede på, hvilke klausuler databasen (dvs. programmet) til hvert tidspunkt måtte indeholde.

Opgave 7.3 Overvej effekten af at lade Lispfortolkeren beskrevet ovenfor udregne værdien af følgende udtryk.

```
[defun, defun, defun, defun]
```

Opgave 7.4 Denne opgave handler om uklarhederne omkring brug af variable inde fra funktioner. Betragt følgende udtryk.

```
[progn, [defun, fup, x, [setq,x, [plus,x,1] ] ],
        [fup, 5]]
```

Hvilken værdi vil man forvente udfra beskrivelsen i afsnit 7.2, og hvordan kan forvente, at fortolkeren beskrevet i afsnit 7.4 vil reagere? Overvej en præcicering af beskrivelsen i afsnit 7.2, så den specificerer en anstændig opførsel. (Advarsel, der findes ikke en entydig besvarelse af denne opgave!)

Opgave 7.5 De fleste Lispdialekter indeholder følgende funktioner.

```
[atom, X], [listp, X], [numberp, X]
```

Værdien af X findes, og det undersøges, om denne er et atom, respektiv en ikke-tom liste, respektiv et tal. Hvis resultatet af undersøgelsen er positivt, er værdien af hele udtrykket »t«, ellers »[]«. Udvid fortolkeren med disse funktioner.

Opgave 7.6 De fleste Lispdialekter indeholder en funktion med navnet »eval«.

```
[eval, X]
```

Betydningen er som følger. Udregn X, betragt den fundne værdi som et nyt udtryk og udregn værdien af dette. Denne facilitet virker i første omgang lidt underlig, men den kan bruges til (f.eks.) at lave funktioner, som definerer andre funktioner med. Et lille, noget enklere eksempel:

```
[setq, a, [quote, [plus, [times, 2, 3], 117]]]]

[eval, [car, [cdr, a]]]
```

Udregnes disse værdier i rækkefølge, forventes værdien af det sidste at være 6. Udvid fortolkeren med »eval«-funktionen.

Opgave 7.7 Opgaven går ud på at udvikle fortolkeren frem til noget, som fremstår som et kørende Lispsystem. Dvs. man skal ikke som bruger kalde noget prædikat, »systemet« indlæser et Lisp-udtryk, evaluerer dets værdi og udskriver resultater — og det bliver det ved med, til man kalder en funktion `[stop]`. Eksempelvis kunne man forestille sig følgende dialog; det understregede er systemets respons, det øvrige brugerens indtastninger.

```
?- run_lisp.   % vi beder Prolog om at starte Lispsystemet.
> [plus, 1, 2]
    3
> [defun, kvadrat, n, [times, n, n]]
   kvadrat
> [kvadrat, 7]
   49
> [stop]
?- ...
```

7.3 Parameteroverførsel

Når vi kalder en funktion, det være sig en indbygget eller en, vi selv har skabt ved brug af `defun`, sker der en overførsel af parametrene til funktionskroppen. Betragt egen vor definition af semantikken for `plus`-funktionen ved følgende Prologregel.

```
lisp( [plus, X, Y], Værdi):-
    lisp(X, Xværdi),
    lisp(Y, Yværdi),
    Værdi is Xværdi + Yværdi.
```

Som vi forventer, evalueres argumenternes værdier altid netop én gang, hvorefter disse værdier gøres til gentand for en addition. Dette kaldes parameteroverførsel ved call-by-value.

Det forholder sig anderledes med `if`, her evalueres betingelsen `B` stadig som call-by-value, de to grene (refereret til ved `X` og `Y`) evalueres ikke ubetinget. Faktisk evalueres altid kun netop en af dem afhængigt af, hvad evalueringen af betingelsen resulterede i.

```
lisp( [if, B, X, Y], Værdi):-
    lisp(B, Bværdi),
    ( Bværdi = [] -> lisp(Y, Værdi)
      ;
      lisp(X, Værdi) ).
```

Dvs. parametrene overføres som »kodestykker«, som kan aktiveres, såfremt den kaldte funktion finder dette fornødent. Dette kaldes parameteroverførsel ved call-by-name.

Da sproget Algol60 blev designet, var det (ved et uheld?) udstyret med call-by-name til overførsel af almindelige procedureparametre, men call-by-name er gået noget af mode siden. Som man kan forestille sig, er det en meget slagkraftig mekanisme, som kan bruges til at skrive meget kompakte programmer. Call-by-name er dog i de fleste tilfælde ikke så effektivt at implementere som call-by-value, og samtidigt kan programmer som for alvor udnytter call-by-name fremstå temmeligt kryptisk.

7.4 En lille diskurs om programtransformation

Programtransformation handler at udføre forskellige manipulationer ved et program med det formål at opnå et bedre program, vel og mærket uden at semantikken ændres. Det kan f.eks. dreje sig om at forbedre en procedures tids- og/eller pladsforbrug. Det er ofte noget, man indbygger i en oversætter, således at man først oversætter til en eller anden mellemkode, man kan udføre transformationer på, og så derefter oversætter til maskinsprog.

I det følgende benyttes Lispfortolkeren skrevet i Prolog beskrevet i dette kapitel til at illustrere fænomenet programtransformation. Til sidst vises et lille eksperiment foretaget med en Pascaloversætter, hvoraf det må konkluderes, at den ikke indholder den slags optimeringer.

7.4.1 Programtransformation i oversættelse fra Lisp til Prolog

Måden, den omtalte fortolker behandler en funktionsdefinition i Lisp på, er at oversætte den til en Prologregel, som så hægtes ind i fortolkeren.

Betragt f.eks. følgende kald af Lispfortolkeren, som behandler en definition af en kvadrat-funktion forfattet i Lisp.

```
?- lisp([defun, kvadrat, n, [times,n,n]], K)

K = kvadrat
```

Vha. `listing`-prædikatet kan vi se, at funktionen er blevet oversat til følgende Prologregel.

```
lisp([kvadrat, A], B) :-
  !,
  lisp([setq, n, A], C),
  lisp([times, n, n], B),
  retract((lisp(n, C):-!)).
```

Den forekommer ikke særlig effektiv, idet den indeholder en hel del ting, som ikke synes nødvendige for det at kvadrere.

Vi kan genkende den generelle parameteroverførselsmekanisme, nemlig at der `setq`'es en variabel, som lever mens kroppen udregnes, og som `retract`es bagefter. Desuden indeholder reglen et par rekursive kald af fortolkeren svarende til funktionskroppen.

Lad os betragte delmålet `lisp([setq, n, A], C)`, som vi vil kalde M1. Når det kaldes, er det en helt bestemt regel i fortolkeren, som aktiveres, nemlig denne her, vi vil kalde Ksetq:

```
lisp( [setq, V, X], Xværdi):-
        lisp(X, Xværdi),
        asserta( (
                  lisp( V, Xværdi):- !
                ) ).
```

Vi kan foretage en *udfoldning* af M1 ved at erstatte den med en kopi af kroppen af reglen, hvor vi har erstattet parametrene med de aktuelle argumenter, dvs. erstattet `V` med `n`, `X` med `A` og `Xværdi` med `C`. Det giver os et nyt mål, M1' =

```
lisp(A, C),
asserta( (
         lisp( n, C):- !
       ) )
```

Det må stå os frit for at benytte M1 eller M1' i en given sammenhæng, da udfoldning af et kald er en operation, som bevarer semantikken. (Her under forudsætning, at der netop er én klausul, som kan anvendes på det givne kald). Havde der været nogle variable, som forekom i Ksetq's krop, men ikke i hovedet, skal disse ombenævnes, hvis de faldt sammen med variable i den kontekst, M1 optræder. Men det er der ikke problemer med her.

På tilsvarende vis kan vi udfolde M2 = `lisp([times, n, n], B)` til M2' =

```
lisp(n, Xværdi),
lisp(n, Yværdi),
B is Xværdi * Yværdi
```

Ved at udskifte M1 med M1' og M2 med M2', opnår vi følgende klausul.

```
lisp([kvadrat, A], B) :-
  !,
    lisp(A, C),
    asserta( (
             lisp( n, C):- !
           )  )
    lisp(n, Xværdi),
    lisp(n, Yværdi),
    B is Xværdi * Yværdi
    retract((lisp(n, C):-!)).
```

Næste skridt i optimeringen er baseret på en *dataflow-analyse* af klausulen, som går ud på at holde styr på, hvilke variable, som indeholder hvilke værdier.

Vi bemærker, at den indledende `assert` angående variablen `n` har den effekt, at umiddelbart efterfølgende udregninger af `n`'s værdi vil give et resultat svarende til værdien af `C`. Vi kan altså forenkle yderligere til følgende:

```
lisp([kvadrat, A], B) :-
  !,
    lisp(A, C),
    asserta( (
             lisp( n, C):- !
           )  )
    Xværdi=C,
    Yværdi=C,
    B is Xværdi * Yværdi
    retract((lisp(n, C):-!)).
```

Yderligere, med vort kendskab til logiske variable og unification, kan vi afskaffe de to »ligninger« ved, så at sige, at udføre dem på oversættelsestidspunktet.

```
lisp([kvadrat, A], B) :-
  !,
    lisp(A, C),
    asserta( (
             lisp( n, C):- !
           )  )
    B is C * C
    retract((lisp(n, C):-!)).
```

Endelig observeres det, at Lispvariablen `n` nu ikke har indflydelse på beregningen af resultatet `B`, og da endvidere bindingen slettes til sidst, så hvorfor i det hele taget oprette den? Altså når vi frem til følgende:

```
lisp([kvadrat, A], B) :-
  !,
    lisp(A, C),
    B is C * C.
```

Opsummering: Vi har ved ligefremme transformationer, som ligeså godt kunne have været udført automatisk, nået til en specialiseret klausul for den nye kvadrat-funktion, som er optimal i den forstand, at den svarer til det, en erfaren Prologprogrammør ville producere ved håndkraft.

7.4.2 Et eksempel på en ikke-optimerende oversætter

I en Pascalversion, hvis navn ikke skal afsløres her, her der udført et lille eksperiment med tidsmålinger for at se, om oversætteren optimerede så godt, som jeg ville forvente.

Vi definerer en datastruktur, som minder om en hægtet liste sådan her.

```
type
  Pterm = ^term;


  term = record
    case tt : term_type of
         variable: (ref: Pterm);
         structure: ( ..... );
         number: ( ..... );
   end;
```

(Faktisk er det implementationen af Prologtermer, i en implementation af Prolog, men det er ligegyldigt for pointen her.)

Til den aktuelle anvendelse er der brug for en `deref`-funktion, som trævler sig igennem kæder af objekter af type `variabel`, for at finde ud af, hvad det alt i alt peges på. Sådan en plejer man at definere nemt og rekursivt sådan her:

```
function deref(pt: Pterm): Pterm;
begin
  case pt^.tt of
    variable: if pt^.ref = nil then
                deref := pt
              else
                deref := deref(pt^.ref);
    structure, number: deref := pt;
  end
end;
```

For at teste effektiviteten kan vi sætte en variabel `q` til at pege på en liste med 1000 elementer og udførte følgende sætning 1000 gange og tage tid.

```
p:= deref(q);
```

Til sammenligning kan vi gentage eksperimentet, men nu med følgende »håndoptimerede in-line-kode«:

```
p:= q;
while (p^.tt = variable) and (p^.ref <> nil) do
   p:= p^.ref;
```

Den sidste viste sig at køre 10 gange så hurtigt! Man kunne forvente, med en kvalitetsoversætter, at de to eksperimenter burde tage eksakt samme tid.

Opgave 7.8 Gentag dette eksperiment med en eller flere Javaoversættere.

8 Erklæringer, typer og typecheck

En måde at se et såkaldt højere programmeringssprog som Java, Pascal, C o.lign. er som et syntaktisk sukkerlag lagt ovenpå den »rigtige« maskinkode: Formålet med at skrive et program er at få manipuleret nogle bits, og programmeringssproget er blot en bekvem notation for at udtrykke bitmanipulationer.

Man kan også vælge at se programmeringssproget som en matematisk notation i hvilket man på en abstrakt og deklarativ måde kan beskrive beregninger på matematiske størrelser, og i øvrigt få dem implementeret ved brug af en oversætter. Typer i programmeringssprog så som heltal, (efterligninger af) reelle tal, arrays osv. understøtter dette syn og gør programmerne mere læsbare, når man tænker på dem som matematiske modeller. Typebetingelser — og tilhørende typecheck i en oversætter — sikrer, at der ikke anvendes operationer på data (dvs. de bitmønstre, som repræsenterer dem), som er meningsløse, f.eks. at den logiske værdi svarende til »sand« ikke divideres med tallet 17.

Brugerdefinerede typer, som i f.eks. Pascal, og klasser som i Simula og Java, gør det muligt for programmøren at definere sine egne typer implementeret ved vilkårligt komplicerede sammensætninger af de simple matematiske typer, som programmeringssprogene er født med. Derved bliver programmeringssprogene ikke blot modelleringsværktøj for matematiske fænomener, men også for fænomener, som ligger tættere på en praktisk virkelighed.[1]

Vi betragter det som velkendt, at abstraktionsmekanismer (typedefinitioner, klasser, procedurer, metoder, osv.) også er medvirkende til at gøre programmer mere robuste, sikre og nemmere at vedligeholde.

[1] Visse prædikanter for objektorienteret programmering går endda såvidt i deres begejstring, at de omtaler en samling klassedefinitioner som en *fysisk* model af et system i verden, ordet »fysisk« omhyggeligt skrevet i kursiv for ar fremhæve objektorienteret programmering frem for alle andre slags programmering. Denne sprogbrug forekommer nærværende forfatter en smule misvisende.

I dette afsnit forlader vi den pragmatiske side og går vi tættere på typer og erklæringer i programmeringssprog som selvstændige fænomener, der skal beskrives og karakteriseres. Vi viser også, hvordan typecheck fungerer og hvordan det kan modelleres på forholdsvis overskuelig måde vha. den metodik, vi har anvist, med Prolog som metasprog. Det skal her fremhæves, at de metoder, man anvender i realistiske implementationer af programmeringssprog med ikke-trivielle typebegreber som regel også er baseret på unification (som ligger implicit i beskrivelserne udfærdiget i Prolog). Vor brug af symboltabeller svarer også så nogenlunde til den måde symboltabeller anvendes i effektive oversættere eller smarte programeditorer, som holder rede på syntaksen og erklæringerne for programmøren.

Vi holder os her til at betragte generelle programmeringssprog, da de indeholder et rigt udvalgt af erklæringsmekanismer. Men det skal understreges, at begreberne og de skitserede metoder i høj grad er relevant og anvendelige i forhold til brugergrænsefladesprog. Her dukker typer og typecheck dukker op i diverse varianter og forklædninger, f.eks. under betegnelsen feltvalidering.

8.1 Simple typer og arrays

Det at knytte typer til udtryk i programmeringssprog er relevant, når disse udtryk dækker over forskellige domæner af værdier. Eksempelvis plejer konstanten `3.14` at denotere et element af de såkaldt reelle tal og `117` står som regel for et heltal. Nogle operationer i et sprog kan kun anvendes på argumenter tilhørende bestemte domæner, hvilket giver sig udslag i syntaksen ved at en operator kun kan placeres i forbindelse med udtryk af bestemte typer. Som eksempel rummer de fleste programmeringsssprog en operation for heltalsdivision, typisk noteret »`//`«, som kun kan anvendes mellem hele tal, og som resultat giver et helt tal. Plus-operatoren anvendes som regel både for addition af hele tal og af reelle tal.

Opgave 8.1 Kan du give en god begrundelse for, hvorfor man i de fleste programmeringssprog anvender to forskellige operatorer for division af heltal, hhv. reelle ral, men som regel kun den samme operator for addition?

Opgave 8.2 Der er i teksten ovenfor skelnet mellem brugen af betegnelserne »operation« og »operator«, tilsvarende »domæne« og »type«. Giv din egen definition af disse begreber, som relaterer til det givne sprogs syntaks og semantik.

Hvis et sprog har et endeligt antal typer kan man i princippet beskrive syntaksen ved at benytte én nonterminal for hver type, f.eks. én for heltalsudtryk og én for reel-talsudtryk. Det er der flere ulemper ved dette: Man får flere regler i

grammatikken for samme operator, og man kan ikke udtrykke polymorfi eller såkaldt overloading (defineres nedenfor).

I stedet kan vi indkode information om typen af et udtryk — og betingelse for korrekt anvendelse af typer og operatorer ved en *abstrakt fortolkning* af abstrakt syntakstræer, efter helt de samme principper som vi benyttede for at beskrive definerende fortolkere for semantikken af et sprog i afsnit 4.4.2. Vi viser et eksempel på en sådan beskrivelse for et sprog af udtryk med reelle og hele tal, men for øjeblikket uden variable. Typer for konstanter kan udtrykkes således, hvor vi benytter Prolog's indbyggede prædikater `integer` og `float` som en del af vores metasprog til at bestemme konstantsymbolerne i vort aktuelle objektsprog til at være af typerne `integer`, hhv. `real`.

```
udtryk(Tal, integer):-
   integer(Tal).


udtryk(Tal, real):-
   float(Tal).
```

En operator så som heltalsdivision, som kun anvender på en enkelt type, kan karakteriseres direkte som følger.

```
udtryk(U1//U2, integer):-
   udtryk(U1,integer), udtryk(U2,integer).
```

Vdr. en plusoperator, som anvender på både hele og reelle tal, kunne vi skrive to adskilte regler i stil med den ovenfor, eller vi kan understrege, at der i en eller anden forstand er tale om »den samme« operator ved slå dem sammen i én regel sådan her:

```
udtryk(U1+U2,T):-
   udtryk(U1,T), udtryk(U2,T),
   (T=integer ; T=real).
```

En konvention om, at heltalsudtryk kan benyttes alle steder, hvor reeltalsudtryk forventes (men ikke omvendt) udtrykkes på denne måde:

```
udtryk(U,real):- udtryk(U,integer).
```

Opgave 8.3 Giv eksempler på udtryk (repræsenteret som Prolog-termer), som accepteres af prædikatet `udtryk` og identificér typen af alle deludtrykkene,

Vi pauserer en stund ved begrebet *abstrakt* fortolkning. Fortolkningen defineret ved klausulerne ovenfor kaldes abstrakt, idet den ikke udtrykker hele sprogets semantik, men netop en abstraktion i egentlig forstand over denne.

Så hvor den fulde detaljerede semantik udsiger at værdien af »`2+2`« er `4`, værdien af »`2+5`« er `5` osv., så udtrykker fortolkningen her, at summen af to vilkårlige heltal blot er et andet heltal, men hvilke konkrete tal, der er tale om, er abstraheret værk. Abstrakt fortolkning er en generelt metode til programanalyse, hvor man er interesseret i at aflure egenskaber ved programmer, typisk for at kunne oversætte dem effektivt. Typecheck er blot ét eksempel herpå, andre egenskaber, man kan være interesseret i, er hvorvidt et udtryk kan beregnes statisk (dvs. på oversættelsestidspunktet) eller dynamisk, hvilket vil sige, at det refererer til variable som ændrer sig under udførelsen.

En operator som »+« kaldes på engelsk *overloaded*.[2] Overloading betyder, at en operator kan anvendes på forskellige type af udtryk, og at den står for i pricippet forskellige funktioner (operationer). I visse sprog kan man yderligere overloade en operator som plus med helt nye betydninger, når den anvendes på udtryk af brugerdefineret type.

Begrebet *polymorfi* er beslægtet med overloading. men anvendes i det tilfælde, at en operator står for en funktion, som kan anvendes på forskellige typer, eller kan beskrives uafhængigt at, hvilke typer den anvendes på. Et godt eksempel på en polymorf operator er den velkendte tilskrivningssætning (assignment), udstyret med den sædvanlige begrænsning, at variabel og udtryk skal være af samme type.

```
sætning(Var:=Udtryk):-
    variabel(Var, Type),
    udtryk(Udtryk, Type).
```

Fordelen ved en beskrivelse som ovenstående er, at den virker for alle typer, inklusive de endnu ukendte af brugeren definerede typer. Bemærk dog, at tefinitionen her kun er anvendelig, såfremt der findes et antal a priori definerede variable, definerede ved fakta som eksempelvis »`variabel(n,integer)`«; vi vender tilbage til variabelerklæringer senere.

Tilsvarende kan vi gøre det muligt at anvende variable i udtryk på samme måde som var de konstanter af samme type ved følgende regel.

```
udtryk(V, Type):-
    variabel(V, Type).
```

Begreberne polymorfi og overloading er nært beslægtede og forveksles ofte. Specielt bliver overloading af operatorer ofte beskyldt for at være polymorfi,

[2]Der findes ikke en etableret dansk sprogbrug svarende til »overloaded«, en gang i mellem støder man på den direkte oversættelse »overlæsset«, hvilket ikke er særligt elegant, så vi holder os til den engelske betegnelse, som har vundet indpas i nudansk.

hvilket er ret misvisende. I nogle terminologier omtales det, at man kan benytte operatoren »+« som navn på en brugerdefineret funktion, som polymorfi, hvilket er temmeligt misvisende, da der ikke er noget til hinder for at man kan vælge at betegne multiplikation af brugerdefinerede komplekse tal ved »+«!

Et andet eksempel på en velkendt konstruktion af polymorf karakter er arrayopslag. Det, at udbede sig det syvende element af et array, skrives på samme måde og kan forstås på samme måde uanset typen af array'ets indhold. For at beskrive typebetingelser for anvendelser af arrays er det relevant at benytte komplekse termer som angivelse af typer, så eksempelvis et array hvis elementer er af en type t kan defineres at have type `array(`t`)`. Vi benytter her en operator »^« til at angive arrayopslag.

```
udtryk(A^Index, Type):-
    udtryk(A, array(Type)),
    udtryk(Index,integer).
```

Vi kan kalde `array(`$-$`)` en polymorf eller parameteriseret type.

Hvis vi for eksempel har et array a priori defineret ved et faktum `variabel(b, array(array(integer)))` siger definitionen, at `b^7` er et legalt udtryk med type `array(integer)` og at `(b^7)^8` et legalt udtryk af type `integer`. Afhængig af karakteren af det sprog, man er ifærd med at beskrive eller definere, kan man vælge at medtage arraygrænserne som en del af typen eller ej.

8.2 Erklæringer og symboltabeller

Et program i et traditionelt programmeringssprog indeholderer en masse navne på ting og sager defineret af brugeren ved brug af forskellige erklæringsmekanismer rækkende fra simple variableerklæringer til klasser med nedarving, virtuelle procedurere og typeparameterisering. Når vi skal analysere os frem til, hvilken betydning, vi kan tillægge en erklæring, kan vi f.eks. fokusere på sammenhængen mellem den virkelighed brugeren (udvikleren, programmøren, designeren,...) er i færd med at beskrive og de program-interne repræsentationer; de fleste metodiske indføringer til objektorienteret programmering vælger denne synsvinkel. Formålet med en klasse `rumraket` er at give en abstrakt repræsentation af rumtraketter, og hvilke attributter, der er revante afhænger af kontekst for anvendelse. Eksistensen af en sådan klassedefinition gør det muligt at oprette objekter, som repræsenterer abstrakte rumraketter, og vi kommer her ind på forskellen på klassedefinitionen og objekterne.

Vi kan påstå, at en klassedefinition eller en hvilken som helst anden erklæring er af en metasproglig karakter, men samtidig indlejret i sproget. Er-

klæringen af en klasse `rumraket` skaber et nyt sprogligt potentiale. I nogle sprog kan man skrive et udtryk `new(rumraket)` hvis betydning er dannelsen af et objekt og som ville være en syntaksfejl, hvis ikke klassen havde været defineret. Den betydning, vi må tillægge en erklæring ligger altså et uldent sted mellem det vi kalder syntaks og det, vi kalder semantik, og hvor metaforen om en metasproglig konstruktion i sproget selv synes at være relevant. En måde at forstå en erklæring på er som en udvidelse af grammatikken (se f.eks. Christiansen, 1985, 1990). Omend denne synsvinkel kan have en filosofisk og lingvistisk tiltrækningskraft, er den knapt så hensigtsmæssig i forhold til beskrivelse af de ofte komplicerede virkefeltsregler, som dukker op ved design af et programmeringssprog.

Den traditionelle måde at beskrive effekten af erklæringer på er ved hjælp af *symboltabeller*. En erklæring genererer en eller flere indgange i en symboltabel, som beskriver det erklærede, og syntaksregler er så parameteriserede med en symboltabel, der beskriver det aktuelle potentiale inden for de kategorier, som nu engang er bestemt at være »udvidelige« i det givne programmeringssprog.

Lad os som eksempel betragte et lille sprog med kun globale erklæringer og udtryk svarende til de, vi så på i det forrige afsnit.

Vi definerer en type på følgende måde: `integer` og `real` er typer; hvis t er en type, så er `array(`t`)` en type. Typer kan benyttes i erklæringer, hvor der dannes en tilknytning mellem et ubrugt symbol (en »identifier«), der introduceres som en variabel, og så variablens type. Vi kan her nøjes med en symboltabel, som er en liste af par af formen `(`id`,`t`)`. Symboltabellen skal gøres tilgængelig ud i samtlige forgreninger af sætninger og udtryk i programmet, hvilket opnås ved at tilføje et ekstra argument til alle syntaktiske prædikater, som forventes at rumme en symboltabel. Symboltabellen kopieres blot ubeset til alle deltræer, eksempelvis:

```
sætning(Var:=Udtryk, Tabel):-
variabel(Var, Type, Tabel),
udtryk(Udtryk, Type, Tabel).


sætning((S1;S2), Tabel):-
    sætning(S1, Tabel),
    sætning(S2, Tabel).
```

Hvor vi ovenfor antog en regel for hver eneste variabel, må vi nu indføre en regel, som, relativt til en symboltabel, definerer hvad der er en legal variabel af en given type.

```
variabel(Var, Type, Tabel):-
    member((Var,Type), Tabel).
```

Som vi har diskuteret, er betydningen af en erklæringen at opfatte som tilføjelse af en indgang i symboltabellen, hvilket vi kan beskrive præcist ved følgende regel, som også medtager den betingelse, at et symbol ikke må erklæres, hvis det allerede har været benyttet. For at kunne beskrive opdatering af tabellen, forsynes det syntaktiske prædikat med to ekstra argumenter, ét svarende til tabellen før og ét til tabellen efter.[3]

```
erklæring(var(Navn,Type),Tabel,[(Navn,Type)|Tabel]):-
    \+ member((Navn,_),Tabel),
    type(Type).


type(integer).
type(real).
type(array(Type)):- type(Type).
```

Vi mangler endnu at beskrive virkefeltet for en erklæret variabel, hvilket vil sige at vi mangler at beskrive regler for den overordnede programstruktur. Vi definerer som følger, idet vi på nuværende tidspunkt antager læseren har vænnet sig til notation og accepteret vore Prologregler som værende *definerende*.

```
program(program(Erklæring,Sætning)):-
    Tabel0=[(var,illegal),(integer,illegal),(real,illegal)],
    erklæring(Erklæring,Tabel0,Tabel1),
    sætning(Sætning,Tabel1).
```

Det fremgår, at et program består at to dele, en erklæringsdel og en sætningsdel, og at symboltabellen tilvejebragt af erklæringerne er tilgængelig i sætningsdelen. Bemærk starttabellen, som indeholder navne, der ikke må benyttes som variable (såkaldt reserverede ord).

Vi mangler nu blot en regel, som sætter en række erklæringer sammen:

```
erklæring((E1;E2),T1,T3):-
    erklæring(E1,T1,T2),
    erklæring(E2,T2,T3).
```

Samles alle reglerne vist i dette afsnit, samt de få, som er udeladt, har vi altså opnået en fuldstændig syntaktisk beskrivelse af et lille sprog med erklæringer, som medtager alle kontekstafhængige begrænsninger og typebetingelse. Udover at tjene som en definition, kan beskrivelsen (fordi den er udformet i

[3] Vi for enkelthedens skyld undladt et test for, at det benyttede variabelnavn er af en art, vi almindeligvis accepterer som en »identifier.«

et implementeret sprog) bruges til at verificere hvorvidt givne Prologtermer repræsenterer korrekte abstrakte syntakstræer i dette sprog.

Vi påstår at have beskrevet det essentielle ved erklæringer og typer, og erklæringer for mere avancerede mekanismer og virkefeltsregler handler blot om lidt mere tricket manipulation af symboltabellen.

Opgave 8.4 Skriv reglerne ovenfor sammen til et Prologprogram og konstruér og aftest nogle eksempler på termer, der godkendes som korrekte programmer og nogle, som forkastes.

Opgave 8.5 Lav beskrivelsen om, så den (som Prolog-program betragtet) giver succes for ethvert syntakstræ, der opretholder den overordnede kontekstfri del af grammatikken for sproget, men til gengæld udskriver meningsfyldte fejlmeddelelser, hvergang øvrige syntaktisk betingelser overtrædes.

Opgave 8.6 Der er et terminologisk problem, vi ikke har taget stilling til i denne fremstilling. Hvad vil du betegne følgende struktur som i forhold til sproget defineret ovenfor?

```
program(var(x,integer), (x:= 3.14 ; y:=x))
```

»Et program«, »et ikke-program« eller »et ukorrekt program«? Begrund hvilke overvejelser, som kan argumentere for de tre muligheder.

8.3 Procedurer med parametre; virkefelt

Følgende Prolog-term repræsenterer et program i et imperativt sprog med rekursive procedurer. Det aktuelle eksempel er en rekursiv version af quicksort.

```
program(
  (var(n,integer);
   var(a, array(integer,4))),

   proc( qsort, (param(left,integer);param(right,integer)),
         ( var(i,integer); var(j,integer);
           var(x,integer); var(w,integer)),

         ( i:= left; j:= right;
           x:= a^( (left+right)//2) ;
           repeat( (while(a^i<x, i:= i+1) ;
                    while(x<a^j, j:=j-1) ;
                    if(i=<j,
                         (w:=a^i; a^i:= a^j; a^j:= w;
                          i:= i+1; j:= j-1))),
                  % until
                  i > j);
            if( left<j, proc(qsort,(left,j))) ;
            if( i < right, proc(qsort,(i,right))) )),

      ( n:= 4 ;
        a:= [30,10,40,20] ;
        proc(qsort,(1,n));
        write(a)) ).
```

Vi skal ikke her ofre papir på en detaljeret beskrivelse af dette sprogs abstrakte syntaks, men blot analysere os frem til, hvordan symboltabellen kan se ud. For at komme frem til det, kan vi se på kaldet af proceduren i sætningsdelen (dvs. »hovedprogrammet«). Den kan vi se som et eksempel på anvendelse af en regel af formen

```
sætning(proc(Id,Pars),Tabel):- ...
```

Indgangen i symboltabellen skal karakterisere præcist det sproglige potentiale, som gøres tilgængelig via procedureerklæringen, dvs. hvad proceduren hedder og hvor mange argumenter, den har, og af hvilke typer. Hvilke navne, disse parametre måtte have i erklæringen, er ikke relevant her, kun deres typer. I princippet kunne vi forestille os alle kald af proceduren beskrevet ved denne regel.

```
sætning(proc(qsort,(Ud1,Ud2)),Tabel):-
    udtryk(Ud1,integer,Tabel),
    udtryk(Ud2,integer,Tabel).
```

Men vi har besluttet ikke at arbejde med udvidelige grammatikker, så vi må indkode information om mulige procedurekald i datastrukturen kaldet en symboltabel. Følgende synes tilstrækkeligt:

```
(qsort, proc((integer, integer))).
```

Vi kan nu fuldende den generelle regel for procedurekald således:

```
sætning(proc(Id,Pars),Tabel):-
    member((Id,proc(ParTyper),Tabel),
    parameter(Pars,ParTyper,Tabel).


parameter(Par,ParType,Tabel):-
   udtryk(Par,ParType,Tabel).


parameter((Par1,Par2),(T1,T2),Tabel):-
    parameter(Par1,T1,Tabel),
    parameter(Par2,T2,Tabel).
```

Inde i selve procedureerklæringen er der til gengæld brug for at kende parametrenes navne (de såkaldt formelle parametere) og deres type, således at eksempelvis følgende fragment

```
x:= a^( (left+right)//2)
```

kan godkendes som en korrekt sætning. Bemærk, at denne sætning også refererer til den globalt erklærede variabel `a` hvis type må forventes at være repræsenteret ved noget i retning af `array(integer,4)`. Symboltabellen skal altså indeholde beskrivelser af globale variable, formelle parameter og lokale variable, når procedurekroppen analyseres, men disse må ikke være tilgængelige andre steder.

Lad os for øjeblikket antage, alle procedurer kun kan kalde sig selv rekursivt samt procedurer erklæret tidligere i teksten. Reglen for procedure erklæringer kan skitseres på følgende måde.

```
erklæring(proc(Id,FormelleParam,LokaleVar,Krop),T0,T1):-
    \+ member((Id,_),Tabel),
    formel_parameter(FormelleParam,ParTyper,T0,T2),
    T1=[(Id,proc(ParTyper))|T0],
    T3=[(Id,proc(ParTyper))|T2],
    erklæring(LokaleVar,T3,T4),
    sætning(Krop,T4).
```

Uden at fylde detaljerne ind, antager vi, at prædikatet `formel_parameter` genererer en struktur af parametrenes type (eks. `(integer,integer)` som ovenfor), at den udvider symboltabellen på samme måde som normale variabelerklæringer. Vi kan nu se, at virkefeltsreglerne, som vi skitserere ovenfor (dvs. hvilke symboler, der kan benyttes hvor) hvemgår utvetydigt. Prologvariablen `T1` udvider den foregående tabel `T0` med viden om de mulige procedurekald og kun det. Symboltabellen `T4`, som beskriver, hvad der er tilgængeligt inde i kroppen opbygges gradvis, først med viden om de formelle parametres navne og typer (`T2`), dernæst tilføjes i `T3` muligheden for rekursivt kald, hvorefter de lokale variabelerklæringer analyseres og deres indhold tilføjes symboltabellen. Det ses tydeligt at viden om de formelle parametre og lokale variable ikke har nogen mulighed for at »slippe ud« af erklæringen til det globale niveau.

Hvis vi ønsker at gøre det muligt for alle procedurer på samme niveau at kalde hinanden, dvs. både procedurer erklæret før og efter den kaldende, er der grundlæggende to muligheder:

- Vi kan analysere teksten (her: syntakstræet) i to passager, første gang analalyseres kun den del af procedureerklæringerne, som angiver, hvad de hedder, og typerne af deres argumenter. Det giver anledning til en symboltabel, som beskriver de mulige procedurekald, og den kan så benyttes i anden passage. En sådan to-passage analyse er forholdsvis nem at specificere i den ramme, vi har beskrevet, i og med programstrukturen er repræsenteret som et træ.

- Analysere programmet i én passage, svarende til, hvordan en effektiv oversætter kan gøre det. Når der mødes kald af en procedure, som allerede er erklæret, kontrolleres kaldet som beskrevet hidtil; for kald af procedurer, som endnu ikke er erklæret, opsamles information om, hvordan den blev kaldt, og når erklæringen mødes, kontrolleres at de registrerede anvendelserne stemmer overens med erklæringen.

Opgave 8.7 (Svær, men lærerig!) Det skitserede metode til analyse af gensidigt rekursive procedurer i én passage kan beskrives meget elegant vha. unification, således at det er den samme symboltabel, der benyttes til såvel registrering af erklæringer som anvendelser af ikke-erklærede procedurer, og reglerne behøver (stor set) ikke skelne mellem de to tilfælde. Opgave lyder: Konstruér og aftest en sådan syntaksbeskrivelse i Prolog.

I beskrivelserne ovenfor har vi ikke været særligt præcise omkring virkefeltsregler. Virkefeltet (eng.: scope) for en erklæring er de dele af programmet i

hvilket det erklærede emne er kan refereres. Vi har hidtil vist princippet om, hvor lokale og globale erklæringer kan ses, men vi mangler at tage stilling til de tilfælde, hvor der erklæres et emne med et navn, som allerede er benyttet på et ydre niveau. Som vi har beskrevet det hidtil, er noget sådant ikke tilladt og omfattes som en syntaksfejl — men dette er ikke tilfredsstillende, idet programmøren således kan risikere at skulle ændre navnene på lokalt erklærede emner, blot fordi en procedure flyttes fra et sted til et andet i programteksten. Den normale konvention er, at i såfald er det blot den lokale erklæring, som får forrang. Vi betragter et eksempel:

```
program(
  var(n,array(integer)),

   proc( p, param(n,integer),
         var(i,integer),
         i:= n),

   p(n^7) ).
```

Navnet `n` benyttes to gange, som et globalt array og som formel parameter, og det, vi vil forvente, er at forekomsten af `n` i kroppen af procedure `p` refererer til den formelle parameter af type `integer`, hvorimod `n` i hovedprogrammet referer til det globale array.

Vi ser altså, at virkefeltet for det globale array er begrænset af den lokale erklæring, der er så at sige et hul i virkefeltet for den globale erklæring (eng.: hole in the scope). Skulle arrayet være tilgængeligt, dvs. en del af den tilgængelige kontekst inde i proceduren, måtte den formelle parameter omdøbes. For at beskrive dette præcist må vi sørge for at symboltabellen tilgængelig inde i procedure kroppen kun gør det muligt at referere til den lokale `n`. Man kunne godt indrette de regler, som opbygger symboltabellen således, at tilføjelsen af den formelle parameter `n` af type `integer` giver anledning til eksplicit sletning at det tidligere `n` af type `array(integer)`. Det er dog nemmere at opdele tabellen i afsnit, således at hver niveau af erklæringer svarer til et afsnit, i vores Prolog version eksempelvis ved en liste af lister. I eksemplet ovenfor kan man forestille sig, at følgende symboltabel er relevant inde i kroppen af proceduren;

```
[ [(i,integer), (n,integer)],
  [(p,proc(integer)), (n,array(integer)]]
```

Det er her hensigtsmæssigt at indføre nogle hjælpeprædikater til opslag i symboltabellen, der fungerer på den måde, at når der søges efter en indgang for

`n`, så det altid er den, som står *forrest* som returneres. Ved hjælp af denne repræsentation er det nemt at beskrive princippet om, at der ikke må erklæres to emner med samme navn på samme niveau — hvilket svarer til, at der ikke må indsættes samme navn to gange i samme tabelafsnit.

Opgave 8.8 Skriv de hjælpeprædikater, som foretager indsættelse og opslag i en lagdelt symboltabel som beskrevet ovenfor og tilpas de grammatiske regler, så de svarer til de skitserede principper for virkefelt. Hvis du har løst opgave 8.7, kan du tilpasse løsningen herfra, ellers benyt den version som er gengivet i teksten.

Den lagdelte symboltabel er også hensigtmæssig i forhold til en effektiv étpassage analyse programmeret i et traditionalt implementationssprog. Her kan symboltabellen fungere som en stak. Hver gang, man går ind i et nyt niveau tilføjes et stakafsnit, som udfyldes ud fra de lokale erklæringer, og som kan afstakkes igen, når (eksempelvis) den lokale procedurekrop er analyseret.

8.4 Klassedefinitioner

Definitioner af klasser er en tand mere kompliceret at beskrive end erklæringer af procedurer. En klassedefinition fungerer som en slags totrinsraket. Først erklæres klassen, og dens navn kan derefter bruges til at erklære variable hvis type svarer til objekter af den given klasse — og når en sådan variabel erklæres, gøres attributter og metoder tilgængelige for denne variabel. Betragt følgende klassedefinition i et til formålet konstrueret sprog og repræsenteret som et abstrakt syntakstræ:

```
class( rumraket,
       (var(position,array(real,3));
        var(hastighed,array(real,3))),
       procedure(flyt, param(dt,real),
          (position^1:= position^1+hastighed^1*dt;
           position^2:= position^2+hastighed^2*dt;
           position^3:= position^3+hastighed^3*dt)))
```

Der erklæres her en klasse med attributter, som angives på samme måde som variable, og metoder, som genbruger syntaksen for procedureerklæringer beskrevet ovenfor. Den syntaktiske betydning af erklæringen kan beskrives ved at associere navnet `rumraket` i symboltabellen med en lille undertabel, som kan »aktiveres« i påkommende tilfælde. F.eks.:

```
(rumraket,
  class( [(position,array(real,3),
           (hastighed,array(real,3),
           (flyt,proc(real))]  ))
```

Erklæring af variable, som refererer til objekter af klassen `rumraket` kan eksempelvis repræsenteres ved et syntakstræ af formen

```
objekt(ariadne,rumraket).
```

Syntaksreglen, som tillader denne og lignende erklæringer, kan formuleres således:

```
erklæring(objekt(Navn,Klasse),Tabel,[(Navn,Klasse)|Tabel]):-
    \+ member((Navn,_),Tabel),
    member((Klasse,class(_)),Tabel).
```

En smule besværligt, men præcist: En klasse skal være til stede i symboltabellen (dvs. klassen skal være erklæret) for at den kan benyttes, og som før tillader vi ikke samme navn benyttet to gange.

Referencer til attributter i et objekt indgår syntaktisk som variable i sproget, hvor den faktiske type (som jo blev defineret i klassedefinitionen) hentes i den lille symboltabel, som er en del af objektets type:

```
variabel(Objekt.Att, Type, Tabel):-
    variabel(Objekt, class(LilleTabel), Tabel),
    member((Att,Type), LilleTabel).
```

Syntaksen for oprettelse af objekter med den specielle funktion `new` er også lige ud ad landevejen (vi minder om, at det andet argument for prædikatet `udtryk` angiver udtrykkets type):

```
udtryk(new(Klasse), Klasse, Tabel):-
    member((Klasse,class(_)).
```

Denne regel virker sammen med reglen for »`:=`«, således at `ariadne:= new(rumraket)` bliver en legal sætning.

Opgave 8.9 Skitsér et lille Java-agtigt sprog med klassedefinitioner og nedarvning og skriv en abstrakt grammatik med symboltabeller i Prolog efter principperne beskrevet i dette kapitel. Sammelign gerne dine beskrivelser med en referencemanual for Java.

8.5 Andre konteksafhængige aspekter af programmeringssprog

De fleste gængse objektorienterede programmeringssprog, eksempelvis Java, mangler den facilitet at kunne parameterisere klassedefinitioner med andre klasser. Det er f.eks. ganske nyttigt at kunne specificere en gang for alle, hvad

man forstår ved en kø på en generel måde, uden at gøre antagelse om, hvilke slags elementer køen indeholder. Vi antyder her, hvordan dette kan gøres vha. en konkret syntaks opfundet til formålet.

```
class kø(parameter class ELEMENT);
   var ...
   procedure indsæt(e: ELEMENT); ...
   function udtag: ELEMENT; ...
end class
```

Idéen er nu, at man kan benytte navnet `kø` som en operator i erklæringer af køer af forskellige slag som antydet i følgende eksempel.

```
var muh1: kø(rumraket);
var muh2: kø(kø(rumraket));
var x,y: real;
...
x:= muh1.udtag.position^1;
muh2.indsæt(muh1);
y:= muh2.udtag.udtag.position^2;
```

Man kan undre sig over, at et forholdsvis nyt objektorienteret sprog som Java ikke medtager parameterisering med klasser, når man ser på de fordele det giver i forhold til struktureret programmering.

Den interesserede læser opfordres til at konstruere en kontekstafhængig syntaksbeskrivelse for sådanne parameteriserede klasser og deres anvendelse.

Parameteriserede klasser er et eksempel på brugerdefineret polymorfi.

8.6 Kort om Prolog og typer

Vi har tidligere diskuteret pragmatiske fordele og ulemper ved typer i programmeringssprog, og vi vil kort her se på muligheden for at benytte typer i logikprogrammeringssprog. I Prolog klarer vi os ganske udmærket med at benytte den generelle datastruktur af termer. Prologs termer, ligesom Lisps lister, har den umiddelbare fordel, at man kan notere en hvilkensomhelst datastruktur direkte. Hvis vi nu havde benyttet Java som vores beskrivelsessprog i dette kapitel, for at repræsentere syntakstræer, symboltabeller osv. osv., ville teksten blive totalt domineret af klasserklæringer og kald af `new`. Omvendt, hvis vort formål havde været at udvikle et nyt programmeringssprog og designe det ud i detaljer, så det kunne videregives til proffesionelle compilerkonstruktører, var det måske en god idé at benytte en »type« beskrivelse af, hvordan en symboltabel er opbygget — med henblik på at sikre den indre konsistens i specifikationen.

Der findes logikprogrammeringssprog, som er udstyret med typeerklæringer, så man kan angive typer for prædikaternes argumenter og de tilladte strukturer, som funktionssymbolerne kan anvende på. Vi kan nævne sprogene PDC-Prolog, Gödel og Mercury. Sproget Gödel tillader typeparameterisering, eksempelvis kan listeoperatoren defineres som en polymorf operator således at man kun kan danne lister bestående af samme type elementer. men denne type kan til gengæld være vilkårlig.

Interesserede læsere opfordres til at konsultere (Hill, Lloyd, 1994), som beskriver sproget Gödel og samtidig giver en god introduktion til typer i logikprogrammeringssprog

Der har også været en ophedet debat om, hvorvidt den seneste ISO standard for Prolog skulle udvides med typedefinitioner, som i al væsentlighed kan opsummeres som en gentagelse af kendte fordele og ulemper ved typedefinitioner. For den nærværende forfatter synes den rigtige løsning at være nærliggende: Prolog (eller hvad sproget nu skal hedde) bør forsynes med *valgfri* typeerklæringer. Et logikprogram har en veldefineret semantik med og uden typedefinitioner. Det gør det muligt for programmøren at starte med at skrive et logikprogram uden typer, eksperimentere med det, og hvis ellers eksperimenterne falder tilfredsstillende ud, kan man forsyne det med typeerklæringer, hvorved man sandsynligvis opdager nogle fejl i det oprindelige utypede program.

Man her kan forestille sig, at programmeringsomgivelser understøtter en metodisk programudvikling, eksempelvis ved en editor som løbende kontrollerer, at symboler anvendes i overensstemmelse med deres typedefinitioner (hvis der altså er nogen). Den anden vej rundt, så kan en tilpas smart editor også udfra et program uden typedefinitioner inducere et forslag til typedefinitioner på baggrund af den måde, symbolerne er anvendt på i programmet (for skitse af en metode, se Christiansen, 1997).

8.7 Opsummering

Vi har således givet de essentielle brikker til en fuldstændig beskrivelse af en abstrakt syntaks for et objektorienteret sprog, og vi vil her stoppe op og overveje, hvad vi egentlig har fået ud af det.

- Vi har illustreret princippet om præcis specifikation af de kontekstafhængige sider af et sprogs syntaks ved hjælp af en forholdsvis forståelig notation.

– Vi har foretaget et detaljeret studium af den kontekstafhængige syntaks for de vigtigste konstruktioner i traditionelle programmeringssprog.

– Vi har vist en metodik med hjælp af hvilken designere af nye objektorienterede og andre sprog kan eksperimentere med konventioner for erklæringer, virkefelt osv. De parameteriserede klasser, vi så på i afsnit 8.5, er et godt eksempel på en ny konstruktion, som var værd at studere nærmere.

– Vi har ikke berørt nær alle subtile problemer omkring virkefelt, som opstår i forbindelse programmeringssprog med ikke-trivielle erklæringsmekanismer og blokbegreber — men dog anvist en ramme, hvori disse problemer kan karakteriseres og studeres.

9 Relationel Algebra, et databasesprog

De sprog, vi har set på indtil videre, drejede sig om at beskrive beregninger som det overordnede formål, hvor Prolog som en undtagelse også (til dels) kan læses som en logisk specifikation. Relationel algebra er et sprog, hvis formål er at kunne beskrive og operere på større mængder af date, typisk administrative data af den slags, man kunne finde på at proppe ind i et databasesystem. Vi præsenterer her sproget med dets semantik, og viser, hvordan det kan repræsenteres og evalueres i Prolog. Som det vil fremgå, er Relationel Algebra i sin grundsubstans nært beslægtet med Prolog og kan på passende vis forstås som en delmængde af Prolog. Kapitlet her kan også tjene som en første indgang til videre studier indenfor databaser.

9.1 En datamodel

En *datamodel* er en model for repræsentation og manipulation af data, dvs. et oplæg, der foreskriver en struktur, der er velegnet til at gemme data i, og et sprog — f.eks. i form af en samling af operatorer — der er anvendeligt til at operere (i database-terminologi: »manipulere«) på strukturen med (læse, skrive, rette og slette data i strukturen). Den datamodel, vi skal se på her, er en forenkling af den relationelle model.

9.1.1 Strukturen

Data repræsenteres i relationer defineret på følgende måde.

> En *relation af grad* n er en navngivet mængde af n-tupler med navngivne komponenter.

Vi kan notere en relation som en tabel, f.eks:

Person:	Pnr	Navn
	123450001	Peter
	123450002	Pia

Her er noteret en relation med navnet »Person«. Den har graden 2 og de to komponenter er navngivet »Pnr« og »Navn« Mængden består af to 2-tupler: $\{\langle 123450001, \text{Peter}\rangle, \langle 1234560002, \text{Pia}\rangle\}$. Vi kalder en relations komponenter for *attributter*. Relationen ovenfor er således udstyret med de to attributter »Pnr« og »Navn«. »Person«-relationen kan anvendes til registrering af personer med personnummer og navn.

Vi skelner mellem *skemaet* og en *forekomst* for en relation. Skemaet for en relation kan betragtes som erklæringen (i database-terminologi: definitionen) og det angiver blot navnene for relationens attributter. For »Person«-relationen illustreret ovenfor kan vi f.eks. notere skemaet således:

Person(Pnr, Navn)

En forekomst af en relation er et aktuelt indhold af relationen. Forekomster af en relation af grad n er blot mængder af n-tupler. Ovenfor er illustreret en forekomst af »Person«-relationen, nemlig mængden: $\{\langle 123450001, \text{Peter}\rangle, \langle 1234560002, \text{Pia}\rangle\}$. Vi kan registrere Søren i relationen ved at indsætte en tupel, f.eks. $\langle 1234560003, \text{Søren}\rangle$ i »Person«. Herved fås en ny forekomst af »Person« nemlig $\{\langle 123450001, \text{Peter}\rangle, \langle 1234560002, \text{Pia}\rangle, \langle 1234560003, \text{Søren}\rangle\}$.

9.1.2 Sproget

Som sprog til at manipulere med relationer indfører vi en algebra bestående af en række velvalgte operatorer. Hver operator tager en eller to relationer som operander. Resultatet af en operation er altid igen en relation. Derfor kan resultatet altid indgå som en operand til en ny relationelt operation. Sproget består således af udtryk, som vi kalder *relationelle udtryk*, og sproget kaldes *Relationel Algebra*. Algebraen omfatter fem operatorer, som gennemgås i det følgende.

Sædvanlige mængdeoperationer: forenings- og fælles-mængde

Begge involverer to operand-relationer. Operatorerne noteres hhv. som »union« og »intersect«. Resultatet af operationerne er:

R union S:	mængden af tupler, der tilhører enten R eller S
R intersect S:	mængden af tupler, der tilhører både R og S

Til forenings- og fælles-mængde vil det være naturligt at indføre et krav om kompabilitet, f.eks.:

> Operand-relationerne til union, intersect skal have samme grad og sammenfaldende attributnavne.

Hermed vil skemaet for resultatet være veldefineret.

Eksempel

Med følgende relationerne,

R:	A	B	C
	a	b	c
	d	a	f
	c	b	d

S:	A	B	C
	b	g	a
	d	a	f

bliver resultatet af en forenings- og en fælles-mængde hhv.:

R union S:	A	B	C
	a	b	c
	d	a	f
	c	b	d
	b	g	a

R intersect S:	A	B	C
	d	a	f

To specielle relations-operationer: projektion og selektion

Begge involverer kun en operand-relation. Når man betragter en relation som en tabel, svarer disse operationer til hhv. søjle- og række-udvælgelse. En projektion noteres således, hvor R står for et relationelt udtryk.

R[⟨attr-liste⟩],

hvor ⟨attr-liste⟩ specificerer en eller flere af attributter for den relation, som R angiver. En selektion noteres

R where ⟨betingelse⟩,

hvor ⟨betingelse⟩ er af formen ⟨attr⟩⟨op⟩⟨værdi⟩, ⟨attr⟩ er en af R's attributter, ⟨op⟩ er en sammenligningsoperator blandt »<«|»≤«|»=«|»≥«|»>« og ⟨værdi⟩ er en konstant værdi.[1]

For en relation med skemaet R[A,B,C] er resultatet af operationerne illustreret i hhv:

[1] I princippet kunne man tillade vilkårlige logiske betingelser opbygget ud fra vilkårlige udtryk, hvor attributnavnene for R indgår som variable.

R[A,C]: en relation med skema »R[A,C]«(A,C), der indeholder mængden af 2-tupler, der fremkommer når man »fjerner« attributten B fra relationen R

R where B=b: mængden af tupler fra R, hvor værdien for attributten B=b.

Bemærk at ikke kun selektion, men også projektion, kan resultere i en relation med færre tupler end i operand-relationen, idet en forekomst af en relation er en mængde.[2]

Eksempel

Resultaterne af en projektion på attributterne A og C hhv. en selektion af tupler med B=b, på en forekomst af relationen R er illustreret herunder:

R:	A	B	C
	a	b	c
	d	a	f
	c	b	d
	a	g	c

R[A,C]:	A	C
	a	c
	d	f
	c	d

R where B=b:	A	B	C
	a	b	c
	c	b	d

Endnu en speciel og særdeles vigtig relationel operation: »naturlig join«

En »naturlig join« involverer to operand-relationer. Det er med denne operation at man på »naturlig« måde kan sammenstille værdier fra relationer, der »har noget med hinanden at gøre«. Vi definerer først konstruktionen på abstrakt vis, hvorefter der følger et eksempel, som forhåbentlig får brikkerne til at falde på plads hos læseren. For to relationer R og S noteres operationen »R join S«.

Fænomenet »har noget med hinanden at gøre« kan vi gøre mere specifikt ved at formulere det som »har sammenfaldende attributter«, og da vi ikke har defineret et type-begreb for attributter, er vores eneste mulighed for at gøre fænomenet konkret at formulere det som »har en eller flere attributter med navnesammenfald«.

For to relationer med skemaer R[A,B,C] og S[B,C,D] er resultatet af en naturlig join:

[2] Dvs. det giver ikke mening at tale om dubletter.

R join S: en relation med skema "R join S"(A,B,C,D), der indeholder mængden af 4-tupler, der fremkommer som mængden af kombinationer af en tupel fra R og en tupel fra S med sammenfaldende værdier for attributter med navnesammenfald (B og C).

Eksempel

For to relationer R og S med forekomster på hhv. 4 og 3 tupler som vist herunder fås i resultatet af en join 6 ud af 12 mulige »kombinations-tupler«, nemlig de hvor R og S har sammenfaldende værdier for attributterne B og C.

R:	A	B	C
	a	b	c
	d	b	c
	b	b	f

S:	B	C	D
	b	c	d
	b	c	e
	a	d	b

R join S:	A	B	C	D
	a	b	c	d
	a	b	c	e
	d	b	c	d
	d	b	c	e
	c	a	d	b

Her følger et eksempel på anvendelse af join-operatoren til en naturligt forekommende opgave, at sætte bynavne på adresser, hvor der kun er angivet postnummer.

Person:	Navn	Gade	Nr	Postnr
	Peter	Algade	17	4000
	Pia	Torvet	7	4600
	Hans	Byvej	25	4600

By:	Postnr	Bynavn
	4000	Roskilde
	4600	Køge
	3900	Nuuk

Person join By:	Navn	Gade	Nr	Postnr	Bynavn
	Peter	Algade	17	4000	Roskilde
	Pia	Torvet	7	4600	Køge
	Hans	Byvej	25	4600	Køge

9.2 En naiv brug af Prolog som relationel database

Før vi præsenterer en fortolker skrevet i Prolog vil vi fundere lidt over den sammenhæng, der er mellem Prolog og relationelle databaser. I begge arbejder man med relationer, i den relationelle algebra med navngivne attributter i simple domæner (i vor version tal, atomer), hvor man i Prolog arbejder med arbitrære strukturer men uden navngivning af felter.

Ser vi bort fra attributnavne, kan vi oversætte en forekomst af en relation til et prædikat i Prolog. En tabuleret relation kan skrives som en række fakta, og en relation defineret ved et udtryk i algebraen kan oversættes til et prædikat defineret ved en regel. F.eks. repræsenterer de to fakta:

```
person( 1234560001, peter ).
person( 1234560002, pia ).
```

relationen Person i en forekomst med de to tupler

$\langle$123450001, Peter$\rangle$, $\langle$1234560002, Pia$\rangle$.

Vi vil illustrere princippet for oversættelse af de relationelle operatorer i forhold til to relationer R og S repræsenteret ved følgende fakta. og vi har tilsvarende tupler af R og S med hhv. 3 og 2 tupler repræsenteret i følgende fakta:

```
r(a, b, c)
r(d, a, f).
r(c, b, d).


s(b, g, a)
s(d, a, f)
```

Og hvad så med sproget/operationerne? De konstruktioner Prolog stiller til rådighed til formulering af forespørgsler, kan vi nå langt med, når vi har valgt at repræsentere relationer som prædikater. En fællesmængde af relationer R og S omfatter tuplerne, der er indeholdt i både R og S, men dette er jo blot hvad vi (i en lidt speciel tabel-form) får som resultat i en sædvanlig konjunktion af delmål i Prolog:

```
?-  r(X,Y,Z), s(X,Y,Z).
```

For at få hele relationen skrevet ud, skal man så bede om at få samtlige løsninger genereret. Tilsvarende fås en foreningsmængde af R og S som en disjunktion af delmål:

```
?-  r(X,Y,Z) ; s(X,Y,Z).
```

En selektion, f.eks. (R where B=b) hvor B underforstås at være anden komponent, fås ved blot at tilføje selektions-betingelsen som et delmål:

```
?-  r(X,Y,Z), Y=b.
```

En simpel projektion kan opnås ved brug af anonym variabel. Hvis vi til R underforstår skemaet R(A,B,C), så kan projektionen R[A,C] udtrykkes som:

```
?-  r(X,_,Z).
```

Endelig kan en join af to relationer med underforståede skemaer R[A,B,C] og S[B,C,D] udtrykkes i

```
?-  r(X,Y,Z), s(Y,Z,W).
```

Hermed fik vi udtrykt alle 5 operationer. Men hvad så med sammensatte operationer? Med den definerede algebra kan vi f.eks. skrive et join (en sammenstilling) af R og S, projiceret på A, som (R join S)[A], men projektionen kan her ikke realiseres ved brug af anonyme variable, så Prolog-målet:

```
?-  r(X,_,_), s(_,_,_).
```

svarer ikke til denne sammensatte forespørgsel. For at opnå at et resultat af en operation har samme form som operanderne, må en forespørgsel formuleres i den mere generelle form:

```
q(...):-  ....
```

altså som en speciel prædikat-definition. Hermed kan vi f.eks. udtrykke ovennævnte (R join S)[A], som:

```
r_join_s_a(X):-  r(X,Y,Z), s(Y,Z,W).
```

En prædikatdefinition kan imidlertid ikke indgå i et mål eller en forespørgsel i Prolog-forstand, men dette behøver strengt taget ikke at være noget problem — vi kan jo blot først tilføje prædikatet til programmet og derefter spørge på instanserne, som f.eks.:

```
?-  r_join_s_a(X).
```

Vi kan se, at Relationel Algebra svarer i udtrykskraft til en delmængde af ren Prolog, hvor rekursion og strukturer er fjernet. Ren Prolog uden strukturer (men med rekursion) svarer til sproget Datalog, vi har omtalt tidligere. Forskellene mellem Datalog og Relationel Algebra er opsummeret i følgende tabel:

	Datalog	Relationel algebra
sammensætning beskrives ved	logiske variable	variable-fri udtryk
evaluering	ét tupel ad gangen	alle tupler på én gang
beregningskraft	vilkårlig rekursion	ingen rekursion
fundamentale operationer i sprog	generel unification	specialiserede, anvendelses-orienterede operatorer

Det, at evaluering foregår på forskellig måde for de to sprog, kan man godt sige, ikke har noget med sproget semantik at gøre, men er ovre i den pragmatiske afdeling, om hvordan sprogene bruges og hvordan de implementeres. Datalogprogrammer kan også evalueres bottom-up (i modsætning til den sædvanlige top-down, mål-orienterede resolution) analogt til, hvordan relationelle udtryk traditionelt evalueres. Omvendt, så har de fleste relationelle databasesystemer et begreb af en cursor, som kan benyttes, netop når man ønsker at modtage tuplerne i en given relation ét ad gangen,

9.3 En fortolker for relationel algebra

En fortolker for Relationel Algebra kan i princippet implementeres efter helt samme princip, som vi evaluerer algebraen af almindelige regneudtryk. Her ville vi få regler i stil med følgende (hvor vi har set bort fra problemet med fjernelse af dubletter):

```
relation( A union B, AuBTupler):-
    relation(A, ATupler), relation(B, BTupler),
    append(ATupler, BTupler, AuBTupler).
```

Dvs. en rekursiv dekomposition, hvor man omsætter hver syntaktisk operator (her `union`) til en tilsvarende semantisk (her `append`). Vi antog her en forekomst af en relation repræsenteret ved en liste af (repræsentationer for) tupler, hvor prædikatet `relation`s andet argument holder sådanne lister.

Det vil være en forholdsvis nem øvelse at skrive denne fortolker færdig, man skal blot for hver operator i algebraen finde — eller programmere — et prædikat, som svarer dertil (på samme måde, som `append` forholder sig til `union`).

Vi kan imidlertid også opnå en elegant fortolker, hvis vi ændrer dens funktionalitet til den Prologagtige ét-tupel-ad-gangen-stil. Relationelle udtryk fortolkes nu af et prædikat af formen

```
tupel( relationelt-udtryk, tupel).
```

Da visse operationer, så som join og selektion afhænger af de aktuelle attributnavnene, indfører vi et analogt prædikat, som for givet relationelt udtryk evaluerer dets skema repræsenteret som en liste.

Vi kan beskrive nogle grundrelationer (dvs. konstanter i algebraen) ved følgende fakta. Der defineres tre relationer, `kunde` der kan opfattes som et kundekartotek, `ordre`, som angiver hvilke kunder, som har bestilt hvilke varenumre i hvilke antal, og endelig `vare`, der beskriver de eksisterende varenumre, og i hvilken by, de skal leveres fra.

```
skema( kunde, [knr,knavn,kby]).
skema( ordre, [knr,vnr,antal]).
skema( vare, [vnr,vnavn,farve,vby]).


tupel( kunde, [k1,søren,lilleby]).
tupel( kunde, [k2,janne,paars]).
tupel( kunde, [k3,benny,paars]).
tupel( kunde, [k4,conny,lilleby]).
tupel( kunde, [k5,adam,arrested]).
tupel( vare, [v1,nød,rød,lilleby]).
tupel( vare, [v2,banan,grøn,paars]).
tupel( vare, [v3,skrue,blå,rungsted]).
tupel( vare, [v4,skrue,rød,lilleby]).
tupel( vare, [v5,kam,blå,paars]).
tupel( vare, [v6,kniv,rød,lilleby]).
tupel( vare, [v7,kost,gul,rungsted]).
tupel( ordre, [k1,v1,300]).
tupel( ordre, [k1,v2,200]).
tupel( ordre, [k1,v3,400]).
tupel( ordre, [k1,v4,200]).
tupel( ordre, [k1,v5,100]).
tupel( ordre, [k1,v6,100]).
tupel( ordre, [k2,v1,300]).
tupel( ordre, [k2,v2,400]).
tupel( ordre, [k3,v2,200]).
tupel( ordre, [k4,v2,200]).
tupel( ordre, [k4,v4,300]).
tupel( ordre, [k4,v5,400]).
```

Uden yderligere besværgelser kan vi allerede nu stille forespørgsler om disse relationer på normal Prologvis.

```
?- tupel(kunde, T).

  T = [k1, søren, lilleby];
  T = [k2, janne, paars];
  T = [k3, benny, paars];
  T = [k4, conny, lilleby];
  T = [k5, adam, arrested];
  no (more) solutions
```

For at kunne notere sammensatte udtryk, definerer vi en bekvem konkret syntaks vha. Prologs operatorer:

```
:- op(500,yfx,union).
:- op(500,yfx,intersect).
:- op(500,yfx,minus).
:- op(500,yfx,project).
:- op(500,yfx,where).
:- op(500,yfx,join).
```

Der er ikke nogen dybere begrundelse for de valgte prioritetstal og angivelse af associativitet. Skema-prædikatet kan defineres på følgende måde, som ikke kræver de store kommentarer. Prædikaterne `liste_indeholdt` og `att_i_betingelse` er hjælpeprædikater, som der overlades til læseren at gætte sig til hvad betyder, og evt. skrive de få programlinier, som definerer dem. Vi overlader alt vdr. `join` som en øvelse til læseren.[3]

```
skema( R1 union R2, Skema):-
    skema( R1, Skema),
    skema( R2, Skema).


skema( R1 intersect R2, Skema):-
    skema( R1, Skema),
    skema( R2, Skema).


skema( R project AttList, AttList):-
    skema(R, RSkema),
    liste_indeholdt(AttList, RSkema).


skema( R where Betingelse, Skema):-
    skema(R, RSkema),
    att_i_betingelse(Betingelse, Att),
    liste_indeholdt(Att, RSkema).
```

[3] . . . som en matematiker ville kalde triviel, men besværlig.

Udover at evaluere sig frem til de faktiske skemaer, vil disse regler også kontrollere, at skemaerne for to relationer nu også passer til den operation, de udsættes for.

Reglerne for `tupel`-prædikatet er lige ud ad landevejen; bemærk kaldene til `skema`, som i de fleste tilfælde kunne være udeladt, det tester, hvorvidt operationen er lovlig anvendt.

```
upel( R1 union R2, Tup):-
    skema( R1 union R2, _),
    (tupel(R1, Tup) ; tupel(R2, Tup) ).


tupel( R1 intersect R2, Tup):-
    skema( R1 intersect R2, _),
    tupel(R1, Tup),
    tupel(R2, Tup).


tupel( R project AttList, Tup):-
   skema(R, Rskema),
   tupel(R, RTup),
   projicer_tupel(Rskema, AttList, RTup, Tup).


tupel( R where Betingelse, RTup):-
   skema(R, Rskema),
   tupel(R, RTup),
   check_betingelse(Rskema, Betingelse, RTup).
```

De to hjælpeprædikater `projicer_tupel` og `check_betingelse` kan programmeres på ligefrem måde, og for at læseren ikke skal tro vi snyder, så følger koden her til nærmere inspektion.

```
projicer_tupel(Skema, [],_,[]).


projicer_tupel(Skema, [Att|Atts], Forekomst, [E|EE]):-
    vælg_komponent(Skema, Att, Forekomst, E),
    projicer_tupel(Skema, Atts, Forekomst, EE).


check_betingelse(Skema, OP(Att,Værdi), Tupel):-
    vælg_komponent(Skema, Att,Tupel, ObsVærdi),
    OP(ObsVærdi,Værdi).


vælg_komponent([Att|_], Att, [K|_], K):- !.


vælg_komponent([_|Atts], Att, [_|F], K):-
    vælg_komponent(Atts, Att, F, K).
```

Her følger et eksempel på en forespørgsel med tilhørende svar. Intuitivt svare den til »Hvilke varer er i ordre i et antal på 200 eller 400?«.

```
?-tupel( (ordre where (antal=200) project [vnr])
         union (ordre where (antal=400) project [vnr]), T).


T = [v2];
T = [v4];
T = [v2];
T = [v2];
T = [v3];
T = [v2];
T = [v5];
no (more) solutions
```

Som det fremgår her, kan denne fortolker ikke finde ud af at sortere dubletter fra, og det vil faktisk også være besværlig. Frasortering af dubletter er noget, som ikke rigtigt passer sammen med Prologs top-down-evaluering. Hvis dette var et ultimativt krav, var det nemmere at indbygge det i en fortolker, som arbejdede på hele relationsforekomster, f.eks. i form af lister, som vi antydede i starten af dette afsnit.

Opgave 9.1 Hvilket abstrakt syntakstræ svarer det konkrete udtryk `kunde join ordre join vare` til under antagelse af de Prologoperatorer, som er defineret ovenfor? Diskuter, om der er andre og mere hensigtsmæssige måder at definere associativitet og indbyrdes prioritet.

Opgave 9.2 Udvid fortolkeren ovenfor, så den kan håndtere `join`.

Opgave 9.3 Skriv en alternative fortolker for relationel algebra, som arbejde-de på hele relationsforekomster, i form af lister, som antydet i starten af dette afsnit. Overvej evt. fjernelse af dubletter.

Opgave 9.4 Ved hjælp at et par små Prolog-fif, kan man med ganske få linier lægge få ét-tupel-ad-gangen-fortolkeren til at udskrive hele relationen uden man skal trykke semikolon en masse gange, og uden dubletter. Hvordan?

Opgave 9.5 I Datalog (eller Prolog for den sags skyld) kan man ofte komme i tvivl om, hvilke argumenter til et prædikat, som står for hvad,[4] og her kunne man foreslå eksplicitte feltnavne, f.eks. at man skulle starte definitionen af et prædikat på følgende måde:

```
:- schema far( far, søn)
```

[4] Er det nu `peter` som er far til `jens` eller omvendt i `far(peter,jens)`?

Man kan indvende, at denne information kan man da bare skrive som en kommentar. Kan du foreslå nogle måder, hvorpå man kan udnytte denne information i sprogets klausuler, f.eks., ved alternative måde at skrive dem på?

Opgave 9.6 Udvid syntaksen for skemaer og fortolkeren i dette afsnit, så man skal angive type for attributterne, hvor disse typer er valgt blandt følgende `tal`, `atom` og `teksstreng`.

10 Turingmaskinen, en model for beregnelighed

Vi har i de tidligere kapitler set på forskellige programmeringsparadigmer, som adskiller sig ved, hvordan man udtrykker ting. Her vil vi gå tættere på at undersøge, hvad man faktisk *kan* eller snarere *ikke kan* udtrykke i et program, som skal udføres af en datamaskine.

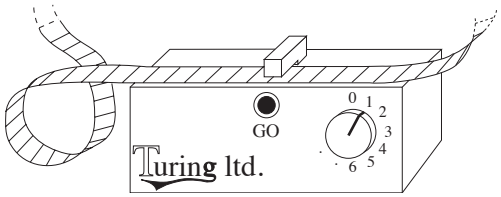
Beskrivelsen af Turingmaskinen er et af de første og mest grundlæggende skridt til etablering af den videnskab, vi i dag betegner datalogi. Alan Turing var i 1930'erne interesseret i begrebet beregninger; hvilke matematiske funktioner er det muligt at beregne og hvilke ikke. For at præcisere dette begreb af beregning definerede han en abstrakt maskine, som kan forstås som en prototypisk datamaskine. Det har senere vist sig, at forskellige andre formelle systemer udviklet med det formål at udtrykke beregnelighed er ækvivalente med Turingmaskiner, f.eks. Church's λ-kalkule (Church, 1941) og, med en passende præcisering, von Neuman-maskinen (Tanenbaum, 1990). Turingmaskinen er først beskrevet i (Turing, 1936); der henvises også til (Dewdney, 1989, kap. 28, 48 og 60; Brady, 1977; Lewis, Papadimitriou, 1981; Penrose, 1989; Sudkamp, 1988). Turingmaskiner er således af stor videnskabshistorisk interesse, men på grund af deres matematiske enkelhed bruges de stadig, når det drejer sig om at afsløre nye resultater om, hvad man kan og ikke kan med en datamaskine.

I afsnit 10.1 forklarer vi indretningen af den simpleste form for Turingmaskiner; i afsnit 10.2 ser vi på forskellige varianter, som er ækvivalente med de simpleste, hvad angår formel udtrykskraft, men som i visse sammenhænge er mere bekvemme. Som et eksempel på udledning af et fundamentalt resultat under brug af Turingmaskiner, viser vi i afsnit 10.3, at det klassiske stop-problem uafgørligt; dette var også beskrevet i Turings oprindelige artikel. Dette resultat er tæt beslægtet med Gödels ufuldstændighedssætning (Gödel, 1931), og der er klare paralleller mellem de respektive beviser.

10.1 Definition og eksempler

Som sagt er en Turingmaskine utroligt enkelt indrettet. Til gengæld er den meget svær at programmere, selv de simpleste opgaver må udtrykkes på meget omstændelig vis – men det gør ikke noget, det, der betyder noget, er det faktum, at opgaverne kan udtrykkes.

En Turingmaskine fungerer som en tilstandsmaskine udstyret med et uendeligt langt bånd, som den læse fra og skrive på. Båndet kan spoles frem og tilbage i ryk svarende til ét felt ad gangen. Vi kan tegne den på følgende måde.



Hvad en given Turingmaskine faktisk foretager sig, er bestemt ved en række transitionsregler, der fungerer som dens program. – Vi definerer nu Turingmaskiner mere præcist, således at de falder indenfor vort begreb af abstrakte maskiner.

Definition. En given Turingmaskine er karakteriseret ved

- et *alfabet* bestående af en endelig mængde af symboler, herunder et specielt *blanktegn.*, som vi noterer »_«,
- en endelig mængde af mulige *tilstande*,
- en endelig mængde af *transitionsregler*, som er fem-tupler af formen,

  $$\langle \textit{aktuel-tilstand, aktuelt-tegn, ny-tilstand, nyt-tegn, retning} \rangle,$$

  hvor *aktuel-tilstand* og *ny-tilstand* er tilstande; *aktuelt-tegn* og *nyt-tegn* tilhører alfabetet; *retning*[1] er en af følgende, »højre«, »venstre«, eller »stille«, hvilket vi noterer ved de respektive symboler $\rightarrow$, $\leftarrow$, og —.

Fælles for alle Turingmaskiner gælder, at

- hukommelsen bestående af følgende:

[1] Bemærk, at retningerne høje og venstre skal forstås sådan, at man rykker hovedet frem og tilbage i forhold til båndet. Hvis man tænker i flytninger af båndet bliver retningerne intuitivt forkerte.

  - et *bånd*, som er uendeligt i begge ender, og som består af felter, som hver rummer ét tegn fra alfabetet,
  - en placering af læse/skrive-hovedet i en specifik position over et felt på båndet,
  - en *aktuel* tilstand.
- det karakteristiske sprog består af endelige strenge over alfabetet,
- den semantiske funktion er bestemt på følgende måde:
  - Den givne input-streng placeres på maskinens bånd således, at resten af båndets felter er blanke; læsehovedet placeres over det tegn i input-strengen, som står længst til venstre, og den aktuelle tilstand sættes til starttilstanden.
  - Sålænge den aktuelle tilstand ikke er en sluttilstand, så
    - vælg en transitionsregel

      $$\langle q_{akt}, t_{akt}, q_{ny}, t_{ny}, retning \rangle$$

      således at q_{akt} svarer til den aktuelle tilstand,[2] t_{akt} til tegnet under læsehovedet.
    - overskriv feltet under læse/skrive-hovedet med t_{ny},
    - sæt den aktuelle tilstand til q_{ny}
    - flyt læse/skrive-hovedet nul eller ét felt i forhold til båndet som *retning* angiver (dvs. hvis *retning* = —, flyttes hovedet ikke).

Bemærk, at den semantiske funktion kan være udefineret i det tilfælde, at maskinen går i uendelig løkke, eller den havner i en ikke-sluttilstand, hvorfra intet træk er muligt. Såfremt maskinen ender i en sluttilstand, siger vi, at den har accepteret den givne input-streng.[3]

Det siger sig selv, at en Turingmaskine er svær at programmere med den ringe struktur, der er på hukommelsen. Har vi eksempelvis brug for noget, som svarer til en variabel, er vi nødt til at placere dens indhold i et eller flere

[2] Der er en tradition i litteraturen for at betegne sådanne tilstande med med bogstavet q forsynet med subskript; begrundelsen herfor eller hvorfra traditionen stammer, henligger i det uvisse.

[3] Man taler ofte om sproget accepteret af en Turingmaskine, hvorved forstås mængden af strenge, den accepterer. Det er helt analogt til terminologien for endelige tilstandsmaskiner og lignende.

felter på båndet. Hver gang dens indhold skal konsulteres eller overskrives, må båndet spoles hen til disse felter. Vi viser her et eksempel på en Turingmaskine, som løser et simpelt tælleproblem. Hvis den gives et startbånd med en sekvens af a'er og b'er, så vil den levere antallet af a'er i et felt umiddelbart til højre for den oprindelige streng. Hvis den f.eks. starter med båndet

$\ldots$, _, _, a, b, a, a, b, b, b, b, _, _, $\ldots$

standser den med et bånd med indholdet

$\ldots$, _, _, b, b, b, b, b, b, b, b, tre, _, _, $\ldots$

Som det fremgår, laver den a'erne om til b'er efterhånden som de tælles (for ikke at komme til at tælle dem med mere end én gang), og dens tælleevne begrænser sig til talsystemet bestående af værdierne nul, en, to, tre, fire, mange.

Alfabet består af mængden { a, b, nul, en, to, tre, fire, mange, _ }, og tilstandende er $q_1, \ldots, q_4$, hvis intuitive betydning er som følger; q_1 er starttilstanden, q_4 den eneste sluttilstand.

q_1: Vi kører hovedet mod højre for at allokere og initialisere et felt som tællevariabel.

q_2: Vi kører hovedet mod venstre i håb om at finde et 'a'.

q_3: Vi kører hovedet mod højre for at opsøge og forøge tællevariablen.

q_4: Slut, alle a'-er er talt med.

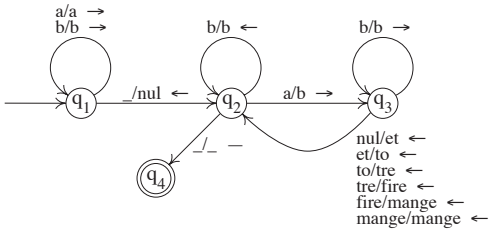
De nødvendige transitionsreglerne er som følger

q_1, a, q_1, a, $\rightarrow$
q_1, b, q_1, b, $\rightarrow$
q_1, _, q_2, nul, $\leftarrow$

q_2, b, q_2, b, $\leftarrow$
q_2, a, q_3, b, $\rightarrow$
q_2, _, q_4, _, —

q_3, b, q_3, b, $\rightarrow$
q_3, nul, q_2, et, $\leftarrow$
q_3, et, q_2, to, $\leftarrow$
q_3, to, q_2, tre, $\leftarrow$
q_3, tre, q_2, fire, $\leftarrow$
q_3, fire, q_2, mange, $\leftarrow$
q_3, mange, q_2, mange, $\leftarrow$

Transitionsreglerne beskrives ofte ved en grafisk notation, hvor man udnytter slægtskabet med de endelige tilstandsmaskiner (kapitel 12). Tilstandene optræder som knuder i en graf, hvor starttilstanden er angivet med en pil (uden startpunkt) pegende på sig, sluttilstanden med dobbelt optrukket cirkel. De øvrige pile svarer til en eller flere transitionsregler, hver angivet ved en etikette, således at en regel $\langle q_{akt}, t_{akt}, q_{ny}, t_{ny}, retning \rangle$ optræder som etiketten $t_{akt}/t_{ny}\ retning$ på en pil, som går fra q_{akt} til q_{ny}. Vi kalder en sådan graf en *transitionsgraf*. Transitionsgrafen svarende til Turingmaskinen ovenfor ser således ud.



Opgave 10.1 Tegn transitionsgrafen for en Turingmaskine, som optæller antallet af a'er i en streng af a'er og b'er, men som ikke sletter a'erne på samme måde som maskinen vist ovenfor. Hvis den f.eks. starter med båndet

…, _, _, a, b, a, a, b, b, b, b, _, _, …

skal den standse med følgende bånd.

…, _, _, a, b, a, a, b, b, b, b, tre, _, _, …

Vink: Tegnsekvensen køres igennem én gang, og den aktuelle tilstand husker, hvor mange a'er der er passeret.

Opgave 10.2 Tegn transitionsgrafen for en Turing-maskine, som, givet en streng af a'er og b'er, ordner disse, så a'erne står samlet og b'erne samlet umiddelbar til højre herfor. Hvis den f.eks. starter med båndet

…, _, _, a, b, a, a, b, b, b, b, _, _, …

skal den standse med følgende bånd.

…, _, _, a, a, a, b, b, b, b, b, _, _, …

Opgave 10.3 Konstruér en Turingmaskine, hvis eneste formål et at dublere sit input, dvs. hvis den gives et startbånd med strengen abcdefgh, så vil den standse med båndet abcdefghabcdefgh.

Der gøres opmærksom på, at eksemplet ovenfor og opgave 10.1 ikke er typiske for den måde, man må repræsentere tal på i en Turingmaskine på. Fordi vi kun havde et endeligt antal talværdier, kunne vi indkode hvert tal som et tegn i alfabetetet og/eller i mængden af tilstande. Alfabetet såvel som mængden af tilstande er pr. definition endelige. Skal vi repræsentere vilkårligt store tal, er vi nødt til at benytte en strategi, hvor hvert tal optager flere felter på båndet. Dette gør så, at programmerne (dvs. transitionsreglerne) bliver temmelig komplicerede – men igen, det interessante ved Turingmaskinerne er deres formelle udtrykskraft.

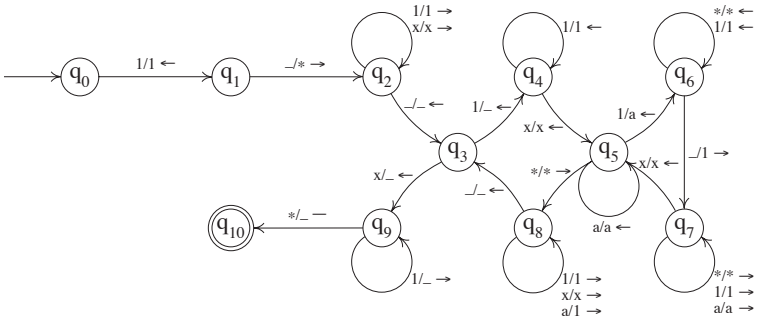
Som afsluttende eksempel viser vi en Turingmaskine, som kan mulitiplicere; den er tilpasset efter et eksempel fra (Dewdney, 1989). Alfabetet består af følgende, $\{1, a, x, *, \_\}$, hvor »x« fungerer som gange-tegn og »*« og »a« er interne hjælpesymboler. Et give tal *n* repræsenteres ved en streng på *n* 1'er, f.eks., 4 repræsenteres ved 1,1,1,1. Hvis startbåndet er

$\ldots, \_, \_, 1, 1, 1, x, 1, 1, \_, \_, \ldots$

standser maskiner med følgende bånd.

$\ldots, \_, \_, 1, 1, 1, 1, 1, 1, \_, \_, \ldots$

Maskinens transitionsgraf er som følger.



Opgave 10.4 Giv en intuitiv beskrivelse af formålet med hver af tilstandende i ovenstående transitionsgraf (svarende til beskrivelsen af den simple tællemaskine beskrevet tidligere).

10.2 Varianter af Turingmaskiner

Vi ser her på forskellige tilpasninger af Turingmaskiner, som gør det mere bekvemt at ræsonnere om forskellige former for beregninger. Fælles for dem alle er dog, at de er ækvivalente i formel udtrykskraft, hvilket vil sige, at

hvis en beregning kan foretages på en maskine T_A, ja så findes der en af type T_B, som foretager samme beregning. — Vi vil blot her antyde, hvordan konstruktioner ser ud og undlade yderligere eksercits ud i formalismens kunst.

Vi kan f.eks. lægge begrænsninger på alfabet eller tilstande, uden det går ud over udtrykskraften.

- Reducere til binært alfabet $\{0,1,\_\}$ mod at der må indføres flere tilstande
- Nøjes med to tilstande mod at alfabetet skal gøres mere kompliceret.

Der kan varieres på båndets format, man f.eks. nøjes med et halvuendeligt bånd. En »halvbåndsmaskine« kan simulere en »helbåndsmaskine« ved at flette de to båndender sammen, så positionerne

$$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$$

optræder i følgende orden på den halvuendelige bånd.

$$0, -1, 1, -2, 2, -3, 3, \ldots$$

Tilstandsmængden og transitionsreglerne skal så modificeres, så båndet rykkes to positioner ad gangen, og der holdes styr på, om det er positionerne med positiv eller negativt indeks, som i øjeblikket er aktuelle.

Simuleringen den anden vej er enkel; en »helbåndsmaskine« kan opføre sig som en »halvbåndsmaskine« ved blot at ignorere halvdelen af sit bånd.

Vi kan også øge antallet af bånd, så en transition nu vil bestå af at sammenholde aktuel tilstand med n tegn (hvor n er antallet af bånd) og skrive n nye tegn, ét på hvert sit bånd, og positionere n læse/skrivehoveder. En flerbåndsmaskine kan simulere en étbåndsmaskine ved blot at benytte ét af sine bånd. Flerbåndsmaskinen simuleres ved at flette båndene sammen og udvide alfabetet til at kunne repræsentere positionerne for de mange hoveder. Vi illustrerer her princippet for en tobåndsmaskine. For hvert tegn x i båndalfabetet tilføjer vi et nyt, $\underline{x}$, med den intuitive betydning, at det svarer til x med læse/skrivehovedet positioneret over sig. Antag f.eks. at de to bånd ser således ud,

$$\ldots, A, B, C, D, E, \ldots$$

$$\ldots, a, b, c, d, e, \ldots$$

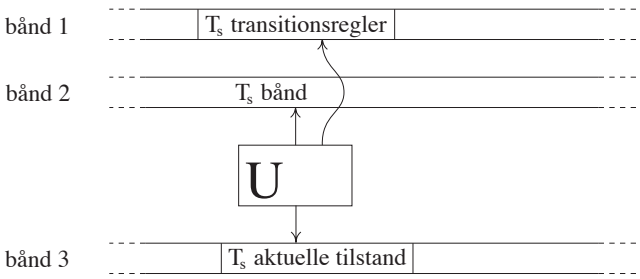
og at de to læsehoveder på en givet tidspunkt står ud for E, respektive c. Det kan repræsenteres ved følgende flettede bånd på étbåndsmaskinen.

. . . , A, a, B, b, C, c, D, d, E, e, . . .

Læse/skrivehovedet for étbåndsmaskinen må så spole frem og tilbage mellem de markeringer, som angiver tobåndsmaskinens hovedpositioner. Transitionsreglerne for étbåndsmaskinen kan fås ved en systematisk omskrivning af tobåndsmaskinens transitionsreglerne.

Den universelle Turingmaskine er specielt interessant ved, at der er tale om én specifik maskine, som er i stand til at simulere alle Turingmaskiner. Med andre ord, en funktion kan beregnes, hvis og kun hvis den kan beregnes på den universelle Turingmaskine. Den universelle Turingmaskine, som vi kalder U, beskrives lettest som en trebåndsmaskine. Antag, vi skal simulere en eller anden Turingmaskine T, så vil Us bånd indeholde følgende:

bånd 1: en indkodning af Ts transitionsregler og angivelse af, hvilke tilstande, som er sluttilstande (dvs. Ts »program«)
bånd 2: en indkodning af Ts bånd (dvs. Ts »data«)
bånd 3: fungerer som en hjælpevariabel, som holder en repræsentation af Ts aktuelle tilstand.



Bemærk, at U arbejder med en *repræsentation* af Ts begreber; hvis U overtog alfabet m.v. fra T, så kunne den jo ikke være universel. U må derfor betjene sig af et universelt båndalfabet, f.eks. $\{0, 1, \sharp, \_\}$, hvor $\sharp$ bruges som et skilletegn. Det vides ikke på forhånd, hvor mange tilstande, T har, og heller ikke, hvor stort Ts alfabet er. Det er altså ikke givet, hvor mange bits, som skal til for at repræsentere tilstand, resp. tegn, men hvis de altid adskilles med $\sharp$, vil Us transitioner kunne identificere hver enkelt tilstand/tegn.

Vi vil ikke konstruere et fuldstændigt sæt transitionsregler for U, men blot nøjes med at skitsere deres overordnede virkemåde.[4]

[4] Vi henregner dette under den kategori, som matematikbøgerne kalder »trivielt, men besværligt.«

0. Hvis den aktuelle tilstand for T (bånd 3) matcher med en sluttilstand, så stop.

1. Der scannes gennem bånd 1 til der identificeres en transitionsregel, som matcher med aktuel T-tilstand (bånd 3) og aktuelt tegn (bånd 2).

2. Hvis en sådan transitionsregel er identificeret, kopieres (bits svarende til) den nye tilstand fra reglen over på bånd 3, (bits svarende til) det nye tegn over i de relevante positioner på bånd 2, hvorefter evt. flytning af Ts hovede afspejles i et antal flyt af hovedet på bånd 2.

3. Gå til pkt. 0.

Hvis man ønsker det, så kan U — som enhver anden flerbåndsmaskine — oversættes til en étbåndsmaskine, som udfører den samme opgave, nemlig at simulere enhver Turingmaskine (incl. sig selv).

Den universelle Turingmaskine er analog til von Neuman-maskinen (se Tanenbaum, 1990), hvad angår det lagrede program. Programmet (bånd 1) er lagret som data, og maskinen fungerer så som en fortolker, der processerer disse data. Med andre ord, hvad der er data og hvad der er program, afhænger af synsvinkel og formål.

10.3 Stop-problemets[5] uafgørlighed

Det forrige afsnit tjente bl.a. til at gøre læseren fortrolig med at jonglere med Turingmaskiner, sådan at argumentationen i dette afsnit ikke skulle volde besvær. Vi betragter følgende centrale, datalogiske spørgsmål.

> Kan man konstruere et program, som afgør, om et vilkårligt andet program vil terminere eller gå i uendelig løkke?

Da vi betragter Turingmaskiner som prototyper på beregnede maskiner eller programmer, omformulerer vi spørgsmålet til følgende.

> Eksisterer der en Turingmaskine T_{stop}, som givet en indkodning af en vilkårlig Turingmaskine T og dennes input X, afgør om T vil standse på X?

Vi vil etablere et klassisk modstridsbevis ved at antage, at T_{stop} eksisterer, og så argumentere frem til, at denne antagelse er absurd. Idéen i beviset er ud fra

[5] Mest kendt under engelske betegnelse »the halting problem«.

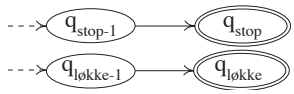
den T_{stop}, vi antog eksistensen af, at konstruere en ny maskine, T_{absurd}, som vi kan anvende på sig selv med absurde resultater.

Vort mål under konstruktionen af beviset er at indrette T_{absurd} sådan at nedenstående spørgsmål hverken kan besvares korrekt med et »ja« eller et »nej«.

Vil T_{absurd} standse på en indkodning af T_{absurd}?

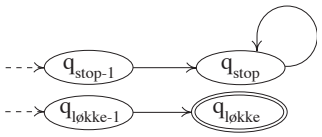
Vi antager altså eksistensen af en Turingmaskine T_{stop}, som beskrevet ovenfor. Uden tab af generalitetet kan vi (jvf. afsnit 10.2) nøjes med at se på stopproblemet for Turingmaskiner T, som benytter et og samme båndalfabet, f.eks. {0, 1, _}. Vi benytter notationen "T" til at betegne indkodningen af T. "T" er altså en beskrivelse af, hvilke tilstande og transitionsregler T indeholder, f.eks. i en binær form. T_{stop} gives altså et input, som indeholder strengen "T" efterfulgt af inputstrengen X, hvilket vi kort vil skrive "T" + X.

T_{stop} vil så — pr. definition — standse på strengen "T" + X og signalere, hvorvidt T ville være standset på X eller ej. Vi kan antage, at T_{stop} signalerer »standser« ved en overgang fra en tilstand $q_{stop\text{-}1}$ til en sluttilstand q_{stop}, og at den signalerer »ikke-standser« ved at gå fra en tilstand $q_{løkke\text{-}1}$ til en sluttilstand $q_{løkke}$.[6]



Pilene skal forstås sådan, at der for ethvert tegn i alfabetet findes en transition mellem de respektive tilstande.

Vi konstruerer nu en ny Turingmaskine T'_{stop} ud fra T_{stop} ved at modificere dens sluttilstande. Vi ænder q_{stop} til at være en ikke-sluttilstand og programmerer T'_{stop} til eksplicit at gå i uendelig løkke i q_{stop}. — Tilstandene $q_{løkke\text{-}1}$ og $q_{løkke}$ ændres ikke.



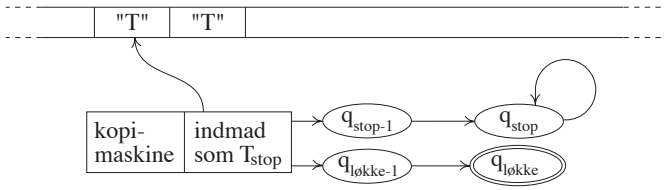
Vi opnår altså, at T'_{stop} går i løkke på de input, som får T_{stop} til at standse og signalere »standser«. For at få modstriden frem, ønsker vi at anvende en maskine med denne opførsel på sig selv, og derfor udvider vi den med en

[6] Hvis ikke T_{stop} opfører sig på præcis denne måde, kan man altid udvide den med nogle få nye tilstande, så den kommer til at gøre det.

kopieringsfacilitet på følgende måde. Vi konstruerer nu maskinen T_{absurd}, således at den først dublerer sit input X, dvs. producerer et bånd med strengen X+X, og derefter opfører den sig præcis som T'_{stop}. (Se opgave 10.3 ovenfor omkring kopieringsfaciliteten).

Med andre ord, T_{absurd} undersøger, hvorvidt en given maskine T vil standse, når T gives en indkodning af sig selv (dvs. "T") som input; hvis dette er tilfældet, så går T_{absurd} selv i løkke i tilstanden q_{stop}. Hvis T ikke standser på "T", så vil T_{absurd} signalere dette ved at standse i tilstanden $q_{løkke}$.

Følgende figur skitserer T_{absurd} umiddelbart efter kopieringsprocessen, hvor dens aktuelle tilstand vil svare til den oprindelige startilstand fra T_{stop}.



Vi opnår vor modstrid gennem betragtning af følgende listige spørgsmål.

Vil T_{absurd} standse på input "T_{absurd}"?

Antag svaret er »ja«. Det et eksempel på et stopproblem, som vi antog T_{stop} var i stand til at afgøre. Dvs. giver vi strengen "T_{absurd}" + "T_{absurd}" som input til T_{stop}, så vil den standse i q_{stop} (jvf. den første tegning). Men T_{absurd} var jo konstrueret ud fra T_{stop}, så vi kan regne ud, hvordan T_{absurd} opfører sig med input "T_{absurd}". Først dublerer den sit input, derefter opfører den sig som T_{stop}, hvorefter den kaster sig ud i den programmerede løkke. Dvs., under antagelse af at T_{absurd} standser på input "T_{absurd}", konkluderer vi at den ikke gør det! Altså duer svaret »ja« ikke.

Vi prøver så, om svaret kan være »nej«. Dvs, vi antager nu, at T_{absurd} ikke standser på input "T_{absurd}". Men det leder os på tilsvarende vis frem til at T_{absurd} standser på input "T_{absurd}"!?!?!

Altså, vi er nået frem til en absurditet, som kun kan skyldes, at den oprindelige antagelse, eksistensen af T_{stop}, var falsk.

Konklusion:

Man kan ikke skrive et program, som generelt opdager uendelige løkker.

Opgave 10.5 Rekonstruér argumenterne i beviset for stop-problemets uafgørlighed, hvor der benyttes Pascalprogrammer i stedet for Turingmaskiner.

10.4 En fortolker for Turingmaskiner i Prolog

Vi har beskrevet Turingmaskiner som formelle indretninger, der repræsenterer essensen af det at kunne beregne. Det essentielle ligger i, at Turingmaskiner er utroligt simple, samtidigt med, at de repræsenterer den (i matematisk forstand) stærkest kendte form for beregnelighed. Med andre ord, hvis noget ikke kan beregnes af en Turingmaskine, så kan det ikke beregnes af nogen kendt form for maskine. Vi viser her en fortolker skrevet i Prolog, som er i stand til at eftergøre beregningerne for en vilkårlig Turingmaskine, når blot dennes transitionsregler repræsenteres som passende Prologfakta.

Det, at vi har skrevet denne fortolker, har to interessante konsekvenser.

- Vi har vist, at enhver beregning, som foretages på en Turingmaskine, kan eftergøres i Prolog. Dvs. Prolog har den samme universelle udtrykskraft[7] som Turingmaskinerne.
- Vi får et værktøj, som kan bruges til at undersøge og afteste Turingmaskiner.[8]

10.4.1 Repræsentation af uendelige bånd i Prolog

En Turingmaskines bånd er uendeligt i begge ender, og det er problematisk i forhold til at opbygge en datastruktur. På den anden side, på et hvilket som helst tidspunkt under kørslen, vil disse båndender altid indeholde lutter blanke tegn, når blot man går tilstrækkeligt langt ud. Det vil sige, på et hvert givet tidspunkt, er der kun endelig meget information at holde styr på, dvs. den »midterste« del af båndet, og hvor de blanke båndender begynder.

Vi benytter Prologatomet `'...'` som en forkortelse for en liste af uendeligt mange blanktegn, og `' '` for et enkelt blanktegn.[9] Disse indgår så i repræsentationen af bånd, idet vi benytter Prologs almindelige listenotation.

[7] Og da min Macintosh er i stand til at udføre Prologprogrammer, så rummer min Macintosh altså samme universelle udtrykskraft som Turingmaskiner (dog begrænset af et antal megabyte RAM).

[8] Vores eksempel-Turingmaskine, som kan multiplicere, har vi overtaget fra (Dewdney, 1989), som formentligt ikke har haft en fortolker til rådighed. Dewdneys originale design viste sig ved aftestning at indeholde to fejl, én overflødig tilstand samt adskillige overflødige transitionsregler.

[9] Vi minder om, at enhver tegnstreng kan fungere som et atom i Prolog, blot er det nødvendigt at omslutte de, som ellers ville forvirre Prologs leksikalske analyse, i enkelte anførselstegn.

Betragt som eksempel et halvuendeligt bånd, som indeholder tegnene d, e, f efterfulgt af uendeligt mange blanktegn. Det kan vi nu repræsentere ved følgende listestruktur i Prolog.

```
[d, e, f | '...']
```

Denne opfattelse af lister svarer ikke til den, der ligger bag Prologs indbyggede prædikater til listehåndtering eller til Prologs »pattern matching« i det hele taget. Vi indfører derfor et nyt prædikat til at konstruere/dekomponere liste-strukturer, som tager højde for den specielle betydning, vi tillægger atomerne `' '` og `'...'`. Betingelsen

`liste(`*hovede*`,` *hale*`,` *liste*`)`

skal være opfyldt netop, hvis *liste* repræsenterer en liste med det angivne *hovede* og *hale*. Prædikatet kan defineres på følgende måde, i det der tages specielt højde for situationer, hvor et blanktegn klippes af/sættes på en i forvejen uendelig liste af blanke; i alle andre tilfælde opfører listestrukturer sig som på sædvanlig vis.

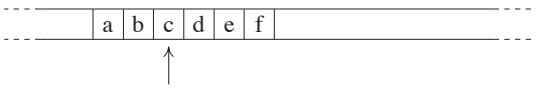
```
liste( ' ', '...', '...'):- !.


liste( Ho, Ha, [Ho | Ha]).
```

Vi kan nu repræsentere et bånd ved en struktur, som følger

`tape(`*venstre*`,` *fokus*`,` *højre*`),`

hvor *fokus* angiver tegnet under læsehovedet, *højre* og *venstre* er listerne svarende til de respektive dele af båndet på hver side af læsehovedet. Det følgende bånd, for eksempel,



er repræsenteret ved følgende struktur, dvs. båndet er så at sige foldet.

```
tape([b,a| '...'], c, [d,e,f| '...'])
```

Vi har nu en datastruktur, som kan repræsentere Turingmaskinens bånd, og vi går nu videre med at implementere de relevante operationer på dem. Det, at aflæse tegnet undet læsehovedet og at overskrive dette tegn med et opdateret bånd som resultat, kan defineres på følgende enkle vis.

```
læs( tape(_, T, _), T).
skriv( T, tape(V_er, _, H_er), tape(V_er, T, H_er)).
```

Det, at flytte læsehovedet i en nærmere specificeret retning, svarer til at sætte tegnet under læse/skrivehovedet over på en af de to lister, og lade den anden listes forreste tegn være det nye tegn under hovedet. For at forenkle fortolkeren angiver vi også en regel, som svarer til at hovedet ikke flyttes.

```
ryk(---, B, B).

ryk(-->, tape( V_er1, F1, H_er1),
         tape( V_er2, F2, H_er2)  ):-
    liste(F1, V_er1, V_er2),
    liste(F2, H_er2, H_er1).

ryk(<--, tape( V_er1, F1, H_er1),
         tape( V_er2, F2, H_er2)  ):-
    liste(F2, V_er2, V_er1),
    liste(F1, H_er1, H_er2).
```

Det anbefales læseren at tegne nogle listemønstre på et stykke papir og indtegne, hvilke lister eller elementer, de enkelte variable svarer til.

Tilsammen bestemmer disse prædikater en abstrakt datatype svarende til Turingmaskiners bånd.

10.4.2 Repræsentation af tilstande og transitioner

En tilstand angives ganske enkelt ved et atom i Prologforstand, og transitionsreglerne udtrykkes som klausuler om et prædikat t. Betragt f.eks. følgende regel taget fra den simple tællemaskine vist i afsnit 10.1.

q_3, nul, q_2, et, $\leftarrow$

Den repræsenterer vi ved følgende faktum.

```
t(q3, nul, q2, et, <--).
```

Bemærk, at Prolog opfatter »pilen« som et atom på lige for med `q3` eller `nul`. På denne måde kan vi indkode alle transitionsregler for den maskine, vi er interesserede i, som et lille Prologprogram bestående af lutter fakta. Følgende eksempel viser hele programmet svarende til tællemaskinen.

```
t(q1, ' ', q2, nul, <--).
t(q1, a, q1, a, -->).
t(q1, b, q1, b, -->).

t(q2, b, q2, b, <--).
```

```
t(q2, a, q3, b, -->).
t(q2, ' ', q4, ' ', --->).

t(q3, b, q3, b, -->).
t(q3, nul, q2, et, <--).
t(q3, et, q2, to, <--).
t(q3, to, q2, tre, <--).
t(q3, tre, q2, fire, <--).
t(q3, fire, q2, mange, <--).
t(q3, mange, q2, mange, <--).
```

Hvilke tilstande, som er start- og sluttilstande, angives ved fakta. For dette eksempel er følgende relevant.

```
start(q1).
slut(q4).
```

Der skal (per. definition) være netop en starttilstand, men der kan dog være flere (fakta om) sluttilstande.

10.4.3 Fortolkeren

Hvis ellers Turingprogrammøren har indkodet sin maskine efter forskrifterne beskrevet ovenfor, er følgende program tilstrækkeligt til at besørge udførelsen; bemærk, at der er to prædikater ved navn `kør`, med to og med fire argumenter.

```
kør( StartBånd, Slutbånd):-
    start(Q),
    kør( StartBånd, Q, Slutbånd, _).

% kør( bånd-før, tilstand-før, bånd-efter,
%                tilstand-efter)

kør( Bånd, Q, Bånd, Q):-
    slut(Q),!.

kør( BåndFør, Qfør, BåndStop, QStop):-
    læs(BåndFør, AktueltTegn),
    t(Qfør, AktueltTegn, Qefter, NytTegn, Retning),
    skriv( NytTegn, BåndFør, BåndMellem ),
    ryk(Retning, BåndMellem, BåndEfter),
    kør( BåndEfter, Qefter, BåndStop, QStop).
```

Iterationen, som vi i et imperativt programmeringssprog ville udtrykke noget i retning af »så længe tilstand ikke er en sluttilstand . . . « er implementeret ved rekursion.[10]

Opgave 10.6 Foretag et håndsimulering af udførelsen af følgende forespørgsel,

```
kør( tape('...', a, [a,b,a,b| '...']), S).
```

idet vi antager, at programmet i afsnit i 10.4.2 er til stede.

10.4.4 Eksempel: Multiplikation med en Turingmaskine

I kapitel 10 viste vi en Turingmaskine, som kunne foretage multiplikation af tal indkodet på passende måde. Vi vil her bruge det til illustrere, hvor effektive Turingmaskiner er. Skrevet om til et Prologprogram efter principperne i afsnit 10.4.2, kommer maskinen til at se sådan ud.

```
t( q0,    1,  q1,    1, <--).
t( q1,  ' ',  q2,    *, -->).
t( q2,    1,  q2,    1, -->).
t( q2,    x,  q2,    x, -->).
t( q2,  ' ',  q3,  ' ', <--).
t( q3,    1,  q4,  ' ', <--).
t( q3,    x,  q9,  ' ', <--).
t( q4,    1,  q4,    1, <--).
t( q4,    x,  q5,    x, <--).
t( q5,    *,  q8,    *, -->).
t( q5,    1,  q6,    a, <--).
t( q5,    a,  q5,    a, <--).
t( q6,    *,  q6,    *, <--).
t( q6,    1,  q6,    1, <--).
t( q6,  ' ',  q7,    1, -->).
t( q7,    *,  q7,    *, -->).
t( q7,    1,  q7,    1, -->).
t( q7,    x,  q5,    x, <--).
t( q7,    a,  q7,    a, -->).
t( q8,    1,  q8,    1, -->).
t( q8,    x,  q8,    x, -->).
t( q8,    a,  q8,    1, -->).
t( q8,  ' ',  q3,  ' ', <--).
t( q9,    1,  q9,  ' ', <--).
t( q9,    *,  q10, ' ', --->).
```

[10]Der er tale om såkaldt halerekursion, da det rekursive kald står til sidst i kroppen af klausulen. Hvis man placerede et »!« uniddelbart før det rekursive kald, vil en god Prologoversætter producere kode, som svarer til den måde en Pascaloversætter ville oversætte en tilsvarende løkkekonstruktion.

Vi vil nu vise samtlige træk, maskinen gennemgår for at gange to med to. Følgende er produceret ved, at der er tilføjet testudskrifter i fortolkeren i afsnit 10.4.3. For hvert skridt udskrives aktuel tilstand sammen med båndets aktuelle indhold. Læse/skrivehovedets position er markeret med parenteser om det element, som er umiddelbart under hovedet.

```
?- kør( tape('...',1,[1,x,1,1|'...'] ), L)


q0:     ..., (1), 1, x, 1, 1, ...
q1:     ..., ( ), 1, 1, x, 1, 1, ...
q2:     ..., *, (1), 1, x, 1, 1, ...
q2:     ..., *, 1, (1), x, 1, 1, ...
q2:     ..., *, 1, 1, (x), 1, 1, ...
q2:     ..., *, 1, 1, x, (1), 1, ...
q2:     ..., *, 1, 1, x, 1, (1), ...
q2:     ..., *, 1, 1, x, 1, 1, ( ), ...
q3:     ..., *, 1, 1, x, 1, (1), ...
q4:     ..., *, 1, 1, x, (1), ...
q4:     ..., *, 1, 1, (x), 1, ...
q5:     ..., *, 1, (1), x, 1, ...
q6:     ..., *, (1), a, x, 1, ...
q6:     ..., (*), 1, a, x, 1, ...
q6:     ..., ( ), *, 1, a, x, 1, ...
q7:     ..., 1, (*), 1, a, x, 1, ...
q7:     ..., 1, *, (1), a, x, 1, ...
q7:     ..., 1, *, 1, (a), x, 1, ...
q7:     ..., 1, *, 1, a, (x), 1, ...
q5:     ..., 1, *, 1, (a), x, 1, ...
q5:     ..., 1, *, (1), a, x, 1, ...
q6:     ..., 1, (*), a, a, x, 1, ...
q6:     ..., (1), *, a, a, x, 1, ...
q6:     ..., ( ), 1, *, a, a, x, 1, ...
q7:     ..., 1, (1), *, a, a, x, 1, ...
q7:     ..., 1, 1, (*), a, a, x, 1, ...
q7:     ..., 1, 1, *, (a), a, x, 1, ...
q7:     ..., 1, 1, *, a, (a), x, 1, ...
q7:     ..., 1, 1, *, a, a, (x), 1, ...
q5:     ..., 1, 1, *, a, (a), x, 1, ...
q5:     ..., 1, 1, *, (a), a, x, 1, ...
q5:     ..., 1, 1, (*), a, a, x, 1, ...
q8:     ..., 1, 1, *, (a), a, x, 1, ...
q8:     ..., 1, 1, *, 1, (a), x, 1, ...
q8:     ..., 1, 1, *, 1, 1, (x), 1, ...
q8:     ..., 1, 1, *, 1, 1, x, (1), ...
q8:     ..., 1, 1, *, 1, 1, x, 1, ( ), ...
q3:     ..., 1, 1, *, 1, 1, x, (1), ...
q4:     ..., 1, 1, *, 1, 1, (x), ...
```

```
q5:     ..., 1, 1, *, 1, (1), x, ...
q6:     ..., 1, 1, *, (1), a, x, ...
q6:     ..., 1, 1, (*), 1, a, x, ...
q6:     ..., 1, (1), *, 1, a, x, ...
q6:     ..., (1), 1, *, 1, a, x, ...
q6:     ..., ( ), 1, 1, *, 1, a, x, ...
q7:     ..., 1, (1), 1, *, 1, a, x, ...
q7:     ..., 1, 1, (1), *, 1, a, x, ...
q7:     ..., 1, 1, 1, (*), 1, a, x, ...
q7:     ..., 1, 1, 1, *, (1), a, x, ...
q7:     ..., 1, 1, 1, *, 1, (a), x, ...
q7:     ..., 1, 1, 1, *, 1, a, (x), ...
q5:     ..., 1, 1, 1, *, 1, (a), x, ...
q5:     ..., 1, 1, 1, *, (1), a, x, ...
q6:     ..., 1, 1, 1, (*), a, a, x, ...
q6:     ..., 1, 1, (1), *, a, a, x, ...
q6:     ..., 1, (1), 1, *, a, a, x, ...
q6:     ..., (1), 1, 1, *, a, a, x, ...
q6:     ..., ( ), 1, 1, 1, *, a, a, x, ...
q7:     ..., 1, (1), 1, 1, *, a, a, x, ...
q7:     ..., 1, 1, (1), 1, *, a, a, x, ...
q7:     ..., 1, 1, 1, (1), *, a, a, x, ...
q7:     ..., 1, 1, 1, 1, (*), a, a, x, ...
q7:     ..., 1, 1, 1, 1, *, (a), a, x, ...
q7:     ..., 1, 1, 1, 1, *, a, (a), x, ...
q7:     ..., 1, 1, 1, 1, *, a, a, (x), ...
q5:     ..., 1, 1, 1, 1, *, a, (a), x, ...
q5:     ..., 1, 1, 1, 1, *, (a), a, x, ...
q5:     ..., 1, 1, 1, 1, (*), a, a, x, ...
q8:     ..., 1, 1, 1, 1, *, (a), a, x, ...
q8:     ..., 1, 1, 1, 1, *, 1, (a), x, ...
q8:     ..., 1, 1, 1, 1, *, 1, 1, (x), ...
q8:     ..., 1, 1, 1, 1, *, 1, 1, x, ( ), ...
q3:     ..., 1, 1, 1, 1, *, 1, 1, (x), ...
q9:     ..., 1, 1, 1, 1, *, 1, (1), ...
q9:     ..., 1, 1, 1, 1, *, (1), ...
q9:     ..., 1, 1, 1, 1, (*), ...
q10:     ..., 1, 1, 1, 1, ( ), ...

L = tape([1, 1, 1, 1|...], ' ', ...)
```

Opgave 10.7 Udvid fortolkeren beskrevet i dette afsnit, så den bliver til en trebåndsmaskine. Husk på, at transitionsreglerne skal udvides, så de beskriver læsning-og-skrivning samtidigt på alle tre bånd. Skriv dernæst et multiplikationsprogram til denne maskine, således at de to tal, som skal ganges, står på bånd 1 og 2; resultatet leveres på bånd 3, som forventes at være blankt fra

begyndelsen. Starter vi eksempelvis med følgende bånd,

bånd 1: ..., 1, 1, 1, ...
bånd 2: ..., 1, 1, ...
bånd 3: ...

så skal maskinene standse med følgende

bånd 3: ..., 1, 1, 1, 1, 1, 1, ...

(Bemærk at »...« står for blanktegn, ikke for at antyde, at 1'erne skal gentages). Vink: Tag ikke udgangspunkt i programmet til étbåndsmaskinen vist ovenfor; programmet til trebåndsmaskinen bliver en hel del enklere.

10.5 Andre uafgørlige problemer

Uafgørlige problemer dukker op overalt i datalogiske problemstillinger og ofte er det tale om problemer, som ved første øjekast forekommer inderligt trivielle. Vi nævner her et nogle eksempler i forbindelse med kontekstfri grammatikker. Lad Σ være et alfabet og G, G_1 og G_2 være arbitrære kontekstfri grammatikker over Σ og $L(G)$ mængden af tegnstrenge genereret af G (en delmængde af Σ^*); tilsvarende $L(G_1)$ og $L(G_2)$ Følgende problemer er uafgørlige:

- hvorvidt $L(G) = \Sigma^*$,
- hvorvidt G er tvetydig, dvs. hvorvidt der findes en tegnstreng, som kan genereres ud fra mere end et syntakstræ,
- hvorvidt $L(G)$ er et regulært sprog,
- hvorvidt $L(G_1) = L(G_2)$,
- hvorvidt $L(G_1) \cap L(G_2) = \emptyset$,
- hvorvidt $L(G_1) \cap L(G_2)$ kan beskrives ved en kontekstfri grammatik.

Der kan f.eks. henvises til (Gruska, 1997), (Dewdney, 1989) eller (Sudkamp, 1988) for beviser og andre eksempler på uafgørlige problemer.

Et andet og tilsyneladende mere subtilt problem er det såkaldte semi-unification-problem. Her viser vi det her i den simpleste form, som kan fremprovokere uafgørlighed. Lad s_1, s_2, t_1 og t_2 være arbitrære termer (i Prolog-forstand). Vi definerer en substitution som en afbildning fra variable til termer. Traditionelt betegnes substititutioner ved græske bogstaver, typisk σ, og hvis

t er en term, noterer $t\sigma$ den term, der fremkommer ved konsekvent at erstatte variable i t med de termer, som σ udsiger. Hvis eksempelvis $\sigma(\texttt{X}) = \texttt{b}$ og $\sigma(\texttt{Y}) = \texttt{g(X)}$ har vi $\texttt{f(a,X,Y,Z)}\sigma = \texttt{f(a,b,g(X),Z)}$. Vi har nu:

- Det er uafgørligt, hvorvidt der findes substitutioner σ, θ_1 og θ_2 så

$$s_1\sigma = (t_1\sigma)\theta_1,$$
$$s_2\sigma = (t_2\sigma)\theta_2.$$

Dette resultat blev opdaget så sent som i 1990 (Kfoury, Tiuryn, J., Urcyczyn, 1990) og er specielt interessant i forhold til typecheck, da det viste, at publicerede og almindeligt anerkendte algoritmer for en specielt form for polymorf typecheck var forkerte.

11 Værktøj i programmeringsomgivelser

For at et programmør kan arbejde effektivt, har han brug for andet og bedre værktøj end blot en flad teksteditor og en compiler. Konstateres der f.eks. fejl i et program, kan det være nyttigt med et værktøj til at kunne følge udførelsen trinvis for at lokalisere fejlen. I forbindelse med optimering af et program kan det også være relevant med et værktøj, som måler, hvilke dele af programmet, der forbruger mest tid. Der findes mange af denne type værktøjer, som afspejler det aktuelle programmeringssprogs semantik, og vi vil her eksemplificere sådanne værktøjer og karakterisere dem vha. metasproglige og udførbare specifikationer. Helt konkret, så tager vi udgangspunkt i fortolkere og oversættere for programmeringssproget Prolog skrevet i metasproget Prolog.

11.1 Tracere og debuggere baseret på Vanilla

Når der foreligger en fortolker i form af et program har man mulighed for at modificere den med tilføjelse af ny funktionalitet. Man kan f.eks. sætte testudskrifter ind, hvorved man får en såkaldt tracer ud af det. For et sprog som Prolog kan man være interesseret i at følge udførelsen ved at se hvilke prædikater, som kaldes, og disses succeser og fiaskoer. En sådan tracer kan fås ved ganske simple ændringer af Vanilla-fortolkeren, vi viste i afsnit 4.5.5, Inden vi viser denne tracer, vil vi indføre et nyttigt kontrolprædikat, som kan skelne mellem om udførelse går forlæns eller der backtrackes.

```
trace_code( Forlæns, Baglæns):-
    vanilla_tracing,!,
    (Forlæns ; Baglæns, fail).


trace_code(_,_).
```

Det parameterløse prædikat `vanilla_tracing` fungerer som et flag, der gør der muligt at slå tracing-faciliteten til og fra (ved `assert`/`retract`).

Den specielle brug af semikolon omkring udskrifterne er et velkendt idiomatisk udtryk blandt erfarne Prologprogrammører, men kan forvirre nybegynderen. Lad $mål_1$ og $mål_2$ være to Prologmål med den egenskab, at de hver især kan generere netop én løsning, dvs. der kan ikke genereres alternative løsninger ved backtracking. Kald af input/outputprædikater har denne egenskab. Betragt da følgende programstump, som udgør sammensat mål.

`(`$mål_1$ `;` $mål_2$`, fail)`

Semikolon betyder som bekendt »eller«, og når det sammensatte mål kaldes første gang, så kaldes $mål_1$, og hele konstruktionen giver succes. Hvis nu der opstår backtracking som følge af, at andre mål fejler, så vil Prologsystemet forsøge at udføre det sammensatte »eller«-mål på en alternativ måde. Det første delmål er afprøvet, og da det antoges kun at kunne generere én løsning, prøver systemet det andet, dvs. $mål_2$`, fail`. Resultatet bliver, at $mål_2$ udføres, hvorefter der fejles, dvs. hele det sammensatte mål fejler.

Vi opsummerer: $mål_1$ udføres under forlæns (optimistisk, søgende, ...) udførelse, $mål_2$ under baglæns (skuffet, backtrackende, ...) udførelse, og det uden at påvirke udførelsesretningen iøvrigt.

Opgave 11.1 Gør rede for, hvilken udskrift, et Prologsystem vil generere, hvis man giver det følgende sammensætning som forespørgsel.

```
( (write(a) ; write(b)) ; (write(c) ; write(d)) ), fail.
```

Det anbefales at afprøve løsningen i Prolog, efter man har fundet den på papir.

Vi benytter også en variation af kontrolprædikatet ovenfor med kun et argument, og hvor det interessante er, at aktionen kun udføres, når tracing er slået til.

```
trace_code( Kode):-
    vanilla_tracing,!,
    Kode.


trace_code(_).
```

For at få de relevante kontroludskrifter genereret, tilpasser vi den tredje klausul i Vanilla, jvf. afsnit 4.5.5 side 68, på følgende måde.

```
solve(A):-
    trace_code( (write('Enter '), write(A), nl),
                (write('Fail '), write(A), nl)),
    clause(A,B),
    trace_code( (write('Try '), write((A:- B)), nl ),
                (write('Drop '), write((A:- B)), nl)),
    solve(B),
    trace_code( (write('Succeed '), write(A), nl)).
```

Under forlæns udførelse udskrives en meddelelse, når et delmål aktiveres (`Enter`), når en objektklausul vælges (`Try`), og endelig, hvis vi når til afslutningen af denne `solve`-klausul, at det pågældede delmål lykkedes. Meddelserne i tilfælde af baglæns udførelse forklarer sig selv.

For at afteste denne tracer benytter vi følgende Prologprogram, som beskriver, hvordan man finder vej i en by med ensrettede gader. Vejkrydsende er betegnet med bogstaver.

```
gadestykke(a, b).
gadestykke(b, c).
gadestykke(c, d).
gadestykke(b, f).
gadestykke(e, f).
gadestykke(f, g).
gadestykke(g, h).


vej(X, Z):-
    gadestykke(X, Z).


vej(X, Z):-
    gadestykke(X, Y),
    vej(Y, Z).
```

Her følger en udskrift produceret af traceren.

```
?- solve( vej(a,f) ).


Enter vej(a,f)
Try vej(a,f):-gadestykke(a,f)
Enter gadestykke(a,f)
Fail gadestykke(a,f)
Drop vej(a,f):-gadestykke(a,f)
Try vej(a,f):-gadestykke(a,_274),vej(_274,f)
Enter gadestykke(a,_274)
Try gadestykke(a,b):-true
Succeed gadestykke(a,b)
Enter vej(b,f)
Try vej(b,f):-gadestykke(b,f)
Enter gadestykke(b,f)
Try gadestykke(b,f):-true
Succeed gadestykke(b,f)
Succeed vej(b,f)
Succeed vej(a,f)
```

En tracer er et eksempel på et programovervågningsværktøj, med hvilket man kan betragte et programs dynamiske egenskaber. Men hvorfor nøjes med at

overvåge? Vi vil her yderligere tillade brugeren at blande sig i udførelsen, så `solve` kommer til at fungere som en såkaldt debugger, hvor brugeren for hvert delmål kan vælge mellem at se detaljeret trace, springe udskrifterne over og gå videre til næste delmål eller bestemme, at målet skal fejle uafhængigt af, hvad programmet måtte mene om den sag. Her følger en ny version af Vanilla, hvor vi har pillet valget af klausul ud og lagt det over i et særligt prædikat, som vi definerer senere. Bemærk, at vi udvider med nye `solve`-regler, så Vanilla nu også kan håndtere primitiverne `fail` og `call(-)`.

```
solve(true):- !.

solve(fail):- !, fail.

solve(call(A)):- !, call(A).

solve((A,B)):- !,
    solve(A),
    solve(B).

solve(A):-
    trace_code( (write('Enter '), write(A), write(?)),
                (write('Fail '), write(A), nl)),
    rewrite_goal_ask_user(A,B),
    solve(B),
    trace_code( (write('Succeed '), write(A), nl)).
```

Bemærk at udskriften ved `Enter` er ændret for at passe sammen med den dialog, vi indfører nedenfor. Et spørgsmålstegn benyttes til at fortælle brugeren, at der forventes en indtastning.

For at kunne implementere den brugerstyrede omskrivning af målet `A` til andre mål `B`, bliver vi nødt til at foretage en indlæsning, hvilket ikke foregår særligt elegant i Prolog. Vi benytter her indbyggede standardprædikater `get0(-)`, som læser en tegnværdi fra tastaturet, og `skip_line` som skipper det linieskift, som brugeren har tastet for at Prolog kan få fat i de(t) forud tastede tegn.[1] En tegnværdi er et tal, f.eks. svarer bogstavet `c` til 99, men for at undgå at skulle huske på hvilke tal, som svarer til hvilke tegn, kan vi benytte en fiks omskrivning, som udnytter Prolog's let barokke måde at opfatte tekststrenge på. En tekststreng skrevet eksempelvis `"abekat"` er intet andet end

[1] Den slags er altid mystisk, og vi vil ikke forklare alle detaljerne. Hvis programmet her benyttes i en anden Prolog version end den, bogens forfatter har benyttet sig, kan det være detaljerne skal justeres en smule.

syntaktisk sukker for en liste af heltalsværdier, i det aktuelle tilfælde `[97, 98, 101, 107, 97, 116]`. Så hvis `X` er en variabel, som holder et tal, vi tænker på som en tegnværdi, kan vi teste om denne værdi svarer til bogstavet `c` ved besværgelsen `[C]="c"`.

Her følger den manglende brik til den nye version af `solve`.

```
rewrite_goal_ask_user(A, B):-
    get0(C), % læse et tegn som talværdi, tomlinie=10
    (C = 10 -> true ; skip_line),


    ( (C=10; [C]="c")
        -> % creep
          clause(A,B),
          trace_code( (write('Try '), write((A:- B)), nl ),
                      (write('Drop '), write((A:- B)), nl))
      ;
       [C]="s"
       -> % skip
          B = call(A)
      ;
       [C] = "f"
       -> % fail
           B = fail
      ;
   write('Du kan skrive tom-linie, c, s eller f?'),
        rewrite_goal_ask_user(A, B)
).
```

Tre kommandoer er tilgængelig for brugeren her, inspireret af et eksisterende Prologsystem.

- `c` for »creep« (kan også angives ved at taste tom-linie) svarende til at dialogen og udskrifterne følger med ind i udførelsen af det aktuelle delmål.
- `s` for »skip« svarende til at det aktuelle delmål udføres uden at detaljerne følges.
- `f` for »fail«; målet fejler.

Her følger et kort eksempel på dialog genereret ved hjælp af den sidste version af Vanilla.

```
?- solve( vej(a,f) ).
Enter vej(a,f)?
Try vej(a,f):-gadestykke(a,f)
Enter gadestykke(a,f)?z
```

```
Du kan skrive tom-linie, c, s eller f?f
Fail gadestykke(a,f)
Drop vej(a,f):-gadestykke(a,f)
Try vej(a,f):-gadestykke(a,_362),vej(_362,f)
Enter gadestykke(a,_362)?
Try gadestykke(a,b):-true
Succeed gadestykke(a,b)
Enter vej(b,f)?s
Succeed vej(b,f)
Succeed vej(a,f)
```

Opgave 11.2 Undersøg debuggeren i det Prologsystem, du almindeligvis benytter og find ud af, hvilke yderligere kommandoer, den tilbyder. Vurdér, hvilke af disse, som umiddelbart kan indføjes i den debuggende Vanilla vist ovenfor, og hvilke, som eventuelt kræver en større omstrukturering. Udvid evt. fortolkeren ovenfor, så den kan håndtere flere kommandoer.

Opgave 11.3 Ovenfor blev Prologs repæsentation af tekststrenge og indlæsningsfaciliteter kritiseret. Er du enig i denne kritik? Foreslå evt. andre måder at designe disse ting til Prolog på.

11.2 Effektivitetsmåling beskrevet ved oversættelse, Prolog $\rightarrow$ Prolog

En såkaldt »profiler« er et værktøj, som holder rede på, hvor mange gange de enkelte programsætninger er blevet udført. I tilfældet Prolog kan det være relevant at holde rede på, hvor mange gange hver enkelt klausul er blevet kaldt, og hvor mange gange disse kald faktisk har ledt til succes. De regler, som kaldes ofte, må forventes at være kritiske i forhold til et programs effektivitet, og i en prioritering af en programmørs opgaver, er det nok sådanne klausuler, han eller hun bør bruge tid på at polere og håndoptimere. Hvis en klausul kaldes ofte, men sjældent giver succes, kunne det være tegn på en uhensigtsmæssig organisering af programmet.

Et sådant værktøj kunne lyde som en meget kompliceret sag at få stablet på benene, men ved at benytte metaprogrammeringsfaciliteterne i Prolog kan det laves forholdsvist enkelt. Man kunne i princippet blot føje det ind i Vanillafortolkeren ved at give den nogle ekstra argumenter, som blev brugt til optællingen. Det er der flere ulemper ved (som også gælder for traceren og debuggeren ovenfor),

- Det er meget ineffektivt, fordi der er det ekstra lag af fortolkning.

– Hvis en fortolker skal kunne bruges til at analysere *vilkårlige* programmer i et givet sprog, så er den nødt til at implementere *hele* dette sprog. Skulle vi udvide Vanillafortolkeren, så den kan klare hele Prolog (med samtlige indbyggede specialiteter, som hører til et givet Prologsystem) ville det være en lang og trælsom proces at få den udviklet.

I stedet bruger vi oversættelse, vi skriver en lille oversætter — i Prolog — som oversætter Prologprogrammer til andre Prologprogrammer, som indeholder ekstra kode, som foretager de optællinger, der er brug for. Denne oversætter er forholdsvis enkel, der skal blot puttes en stump kode ind i starten og i slutningen af kroppen af hver klausul. Hvad kroppen i øvrigt måtte indeholde af sære konstruktioner er ligegyldigt.

Vor idé er at tildele hver klausul et indeks i, og til hvert indeks hører to tællere, én svarende til antallet af gange klausul i er startet og én svarende til antallet af gange klausul i er afsluttet med succes (dvs. er nået til sin slutning). Hvis klausulen `p(X):- q(X)` har indeks 4, skal den erstattes med følgende.

```
p(X):- tæl_en_starter(4), q(X), tæl_en_success(4).
```

De to tælleprædikater påvirker et globalt tælleværk, som i Prolog kan implementeres ved `assert` og `retract`, som manipulerer passende fakta. Hvis, på et givet tidspunkt, klausul 4 er kaldt 27 gange og er afsluttet med succes de 12 af gangene, vil faktummet for klausul 4 se sådan ud.

```
tælle(4, (p(X):- q(X)), 27, 12)
```

Vi opbevarer den originale version af klausulen, så vi kan benytte den i udskrift, og den er også nyttig, hvis tælleriet skal slås fra igen, dvs. den originale version af klausulen kan reetableres.

Det er en ligefrem opgave at definere de to tælleprædikater, og tilsvarende for et prædikat `nyt_index(-)` som, hver gang man kalder det, giver et index, som er én højere end det forrige.

Opgave 11.4 Gør det.

Nu står blot tilbage at definere prædikater, som brugeren kan anvende til at aktivere faciliteten og til at få udskrevet en tabel. Vi definerer et prædikat `profiler` (udtales »profilér«), som kaldes med det prædikat hvis klausuler ønskes målt, f.eks. `profiler(vej(_,_))`. For at de oversatte klausuler kan blive lagt i samme rækkefølge som de oprindelige, fjerner vi først alle relevante klausuler, og tilføjer dem derefter på én gang. Følgende Prologkode er tilstrækkeligt.

```
profiler(Hovede):-
    snup_alle_klausuler(Hovede,Ker),
    dekorer_klausuler(Ker).


snup_alle_klausuler(Hovede,[(KopiHovede:-Krop)|Ker]):-
    copy_term(Hovede, KopiHovede), % indbygget
    retract((KopiHovede:- Krop)), !,
    snup_alle_klausuler(Hovede, Ker).


snup_alle_klausuler(_,[]).


dekorer_klausuler([]).


dekorer_klausuler([(Hovede:-Krop)|Ker]):-
    nyt_indeks(I),
    assertz( taelle(I, (Hovede:-Krop), 0, 0) ),

    assertz( (Hovede :- tael_en_starter(I),
                    Krop,
                    tael_en_success(I) ) ),


    dekorer_klausuler(Ker).
```

Opgave 11.5 Prædikatet `copy_term` er et standardprædikat, som for en givet term producerer en kopi med nye variable. Hvad ville der ske, hvis kaldet af `copy_term` blev udeladt ovenfor, og der efterfølgende stod

```
retract((Hovede:- Krop))?
```

Her følger et eksempel på en måling af et program vist tidligere i dette kapitel; prædikatet `statistik` foretager en passende formatteret udskrift af informationen gemt i `tælle`-fakta'erne.

```
?- profiler(vej(_,_)), profiler(gadestykke(_,_)).
yes
?- vej(a,g), vej(b,h), vej(a,d).
yes
?- statistik.
Klausul: vej(_160,_161):-gadestykke(_160,_155),vej(_155,_161)
kald: 10 succes: 6


Klausul: vej(_154,_155):-gadestykke(_154,_155)
kald: 13 succes: 3
```

```
Klausul: gadestykke(c,d):-true
kald: 3 succes: 3

Klausul: gadestykke(b,c):-true
kald: 3 succes: 3

Klausul: gadestykke(a,b):-true
kald: 2 succes: 2

Klausul: gadestykke(g,h):-true
kald: 1 succes: 1

Klausul: gadestykke(f,g):-true
kald: 2 succes: 2

Klausul: gadestykke(b,f):-true
kald: 2 succes: 2

Klausul: gadestykke(e,f):-true
kald: 0 succes: 0
```

Opgave 11.6 Overvej på baggrund af dette eksempel, om udskriften genereret af `statistik` kunne forbedres, og i givet fald hvordan. Præcisér de kriterier, du anvender her for at vurdere, at noget er en forbedring.

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 ele-

gant.

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.

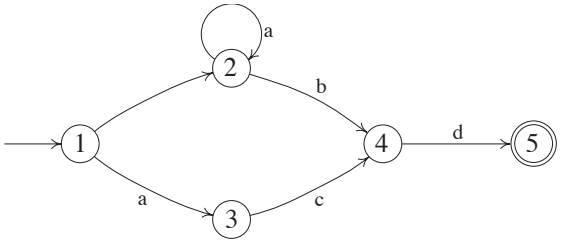
En given tilstandsmaskine beskrives nemmest grafisk ud fra følgende principper.

- 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

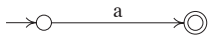
- samme tegn forekommer i etiketten på mere end én pil væk fra tilstanden, eller
- der går mere end én pil væk fra tilstanden, og en af dem har tom etikette.

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

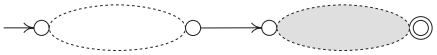
12.2 At lave regulære udtryk om til tilstandsmaskiner og at gøre disse deterministiske

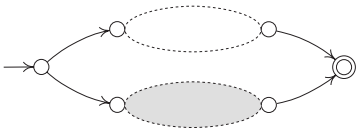
Regulære udtryk kan omskrives til tilstandsmaskiner på en enkel og systematisk måde. Dette kan beskrives som en rekursiv procedure, som går på strukturen af det regulære udtryk, således at hvert deludtryk svarer til sin maskine. Vi kan skitsere proceduren som følger. Konstruktionen foregår således, at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

– ε svarer til 

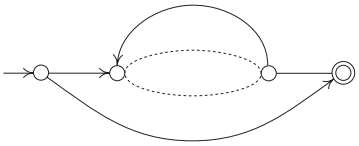
– et tegn *a* svarer til 

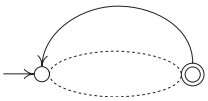
– hvis A svarer til  og B til 

så svarer AB til 

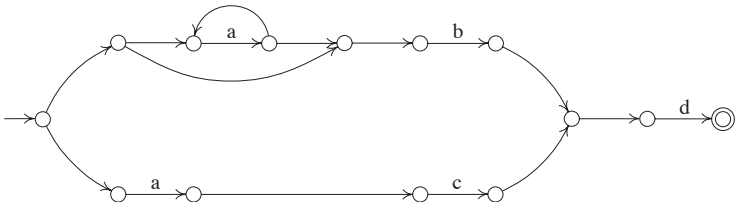
og A+B til 

– 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.

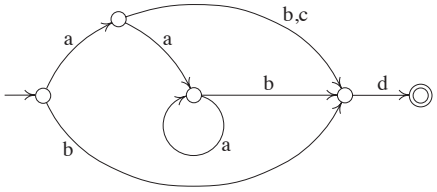


Begrundelsen for indførelse af nye knuder og pile er at undgå problemer, hvis der er løkker tilbage til startknuden/slut-knuderne inde i de tilstandsmaskiner, som sættes sammen. Det kan man nemt overbevise sig om ved at lege lidt med at sammensætte diagrammer for a^* og b^* på forskellig vis.

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,

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

[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 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.

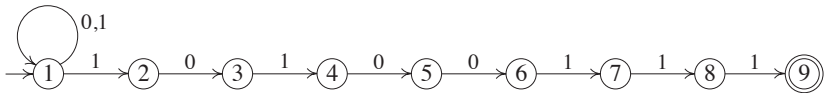
Vi viser princippet vha. et eksempel. Antag, vi søger i strenge af 0'er og 1'er, og at den delstreng, vi leder efter, er 10100111. Følgende tilstandsmaskine vil acceptere netop denne streng og intet andet.

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: ††††

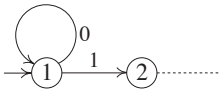
Det program, vi får ud af det, vil altså acceptere samtlige strenge, som begynder med 10100111. Så vidt så godt. Nu vil vi ændre på tilstandsmaskinen, så den også vil identificere søgestrengen, såfremt denne måtte optræde længere inde i teksten. Mere præcist formuleret, vi vil konstruere en tilstandsmaskine,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



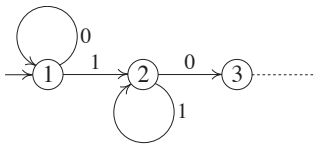
Den kan så køres igennem en algoritme af den slags, vi postulerede i afsnit 12.2, som fjerner ikke-determinisme. Den tilstandsmaskine, vi får ud af det, kan vi så igen køre igennem en generator, som producerer et program i stil med det i afsnit 12.2.2. Så er vi færdige, vi har et specialiseret og effektivt 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

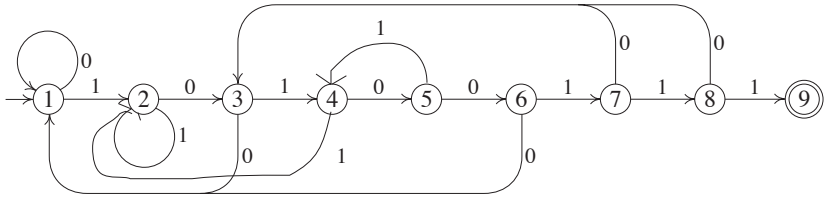


Hvis nu maskinen har konsumeret nul eller flere 0'er, den møder 1 og derefter nok en 1, hvad så? Vor »hypotese« om, at det første 1 angav starten på vor

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 10100111**0**, 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 10<u>101</u>, 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.

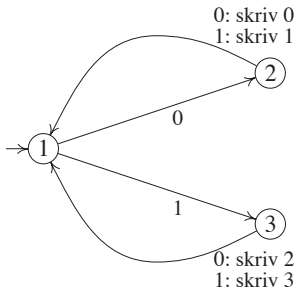
Metoden kan selvfølgelig generaliseres til at lede efter mønstre, som er beskrevet ved et arbitrært regulært udtryk. Analysen og tilpasningen af den tilhørende tilstandsmaskine bliver blot en anelse mere kompliceret. — Men man kan altid henvise til den automatiserede metode.

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 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

[3]Engelsk, udtales måud.

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 »Δ«, linieskift som »¶«.

```
ΔΔbegin¶ΔΔΔΔx:=Δ17
```

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.

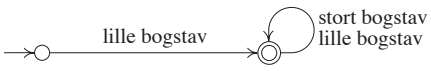
For at benytte tilstandsmaskiner og de implementationsteknikker, vi har udviklet i dette kapitel, skal vi altså

- 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.

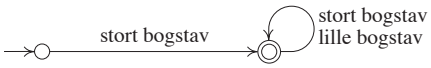
Til det sidste er det ofte relevant også at opsamle den faktiske teksstreng, f.eks. svarende til variabelnavne. Vi benytter her den leksikalske syntaks for Datalog til at illustrere de generelle principper.

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.

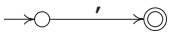
Atomer:



Variable:

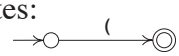


Komma:

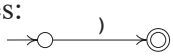


Punktum:

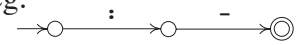
Startparentes:



Slutparentes:



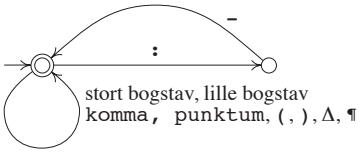
Kolon-streg:



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.

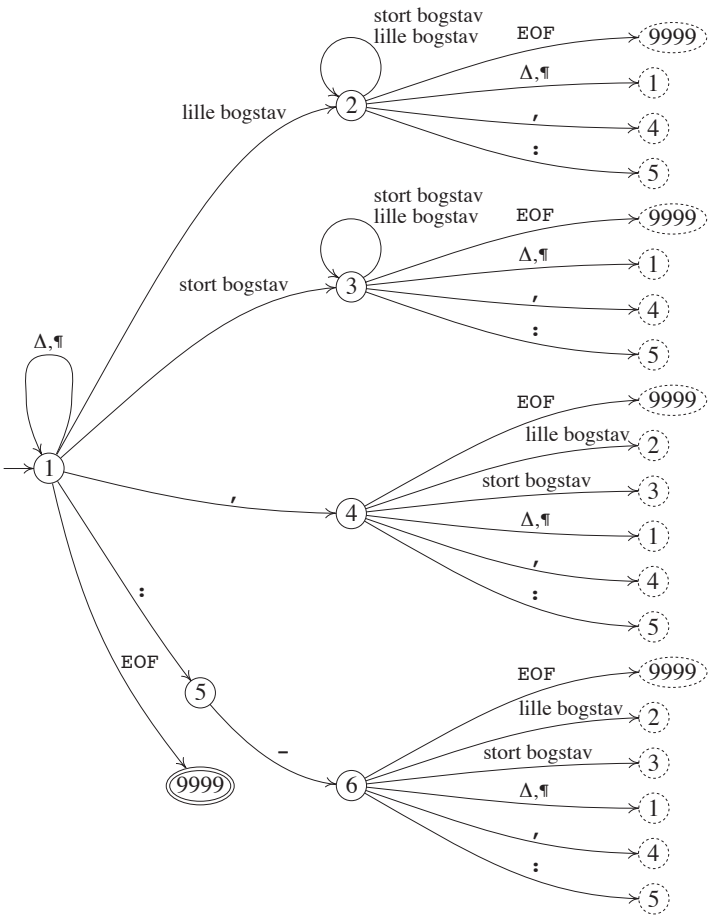
Vi vil i stedet bygge en maskine ved at sammensætte maskinerne for de enkelte symboler, således at

– 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, Var ...`

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, Var ...`

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, Var ...`

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

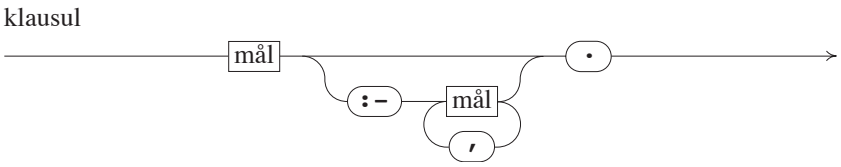
Når de leksikalske symboler i en tekst er identificeret, giver det mening at forsøge at identificere den strukturelle syntaks for derved at kunne rekonstruere de abstrakte syntakstræer, som måtte have eksisteret i hovedet på den programmør eller bruger, som har indtastet den tekst, der er under analyse.

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.

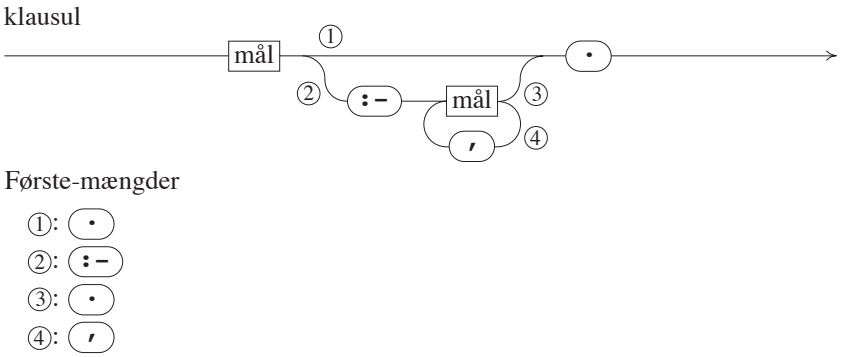
Vi skitserer nu den del af parserens algoritme, som bruges til at indlæse klausuler som følger, idet vi antager at variablen `source` referer til et objekt af klassen `LEXICAL_SCANNER`. Vi benytter her Simula som beskrivelsessprog, men læsere, som er fortrolige med Java, skulle også kunne gennemskue 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]

[1] Eksemplet her er lidt uheldigt fra et pædagogisk synspunkt, da samtlige førstemængder



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:

---

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, slutteligen 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 synboler 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 •. For at få plads på siden forkortes nonterminalerne til et enkelt bogstav.

Stakken	Resterende ind-fil	Handling
•	$t + t * t * t + t$	skift:
• $\underline{t}$	$+ t * t * t + t$	reducer vha. regel 4:
• u	$+ t * t * t + t$	skift:
• $u +$	$t * t * t + t$	skift:
• $u + \underline{t}$	$* t * t + t$	reducer vha. regel 4:
• $u + u$	$* t * t + t$	skift:
• $u + u *$	$t * t + t$	skift:
• $u + u * \underline{t}$	$* t + t$	reducer vha. regel 4:
• $u + \underline{u * u}$	$* t + t$	reducer vha. regel 2:
• $u + u$	$* t + t$	skift:
• $u + u *$	$t + t$	skift:
• $u + u * \underline{t}$	$+ t$	reducer vha. regel 4:
• $u + \underline{u * u}$	$+ t$	reducer vha. regel 2:
• $\underline{u + u}$	$+ t$	reducer vha. regel 1:
• u	$+ t$	skift:
• $u +$	t	skift:
• $u + \underline{t}$		reducer vha. regel 4:
• $\underline{u + u}$		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* ::= $udtryk_1$ $\star$ $udtryk_2$
→ $udtryk_1$ $udtryk_2$ 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.

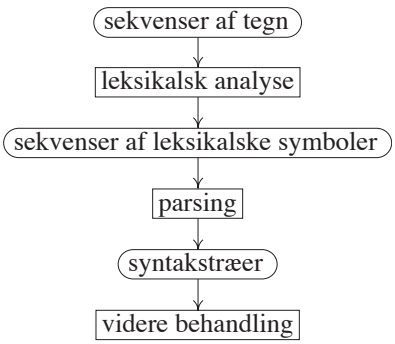
Fordelen ved en tabelstyret implementation (f.eks. Prolog-fakta om de aktuelt definerede operatorer) er, at tabellen og dermed parseren kan udvides »on the flight«, som vi kender det fra Prolog.

14 Syntaksgenkendelse koblet med fortolkning

Vi tidligere vist begreber og metoder omkring syntaks og syntaksanalyse, og her viser vi, hvordan dette kan indgå i en total implementation af et sprog. Syntaks handler om form, og syntaksen kan siges at være implementeret ved repræsentationen af abstrakte syntakstræer og de metoder, som ud fra en kildetekst kan genkende og opbygge syntakstræer.

Semantikken, dvs. betydningen af et sprog kan implementeres ved en oversætter eller en fortolker, og vi viser her et eksempel, hvordan en fortolker kan skrues sammen med de metoder, vi har set til syntaksgenkendelse. Fortolkeren arbejder så på de abstrakte syntakstræer.

Vi kan opsummere den overordnede struktur for systemer, som foretager en eller anden form for sprogbehandling ved følgende figur; afrundede kasser refererer til datastrukturer, de firkantede til processer, som transformerer mellem forskellige repræsentationer.



De enkelte processer behøver ikke udføres sekventielt sådan at en proces skal afsluttes, før den næste får lov til at starte. Ofte vil de fungere som corutiner på den måde, at en overordnet proces (f.eks. parsing) aktiverer en underordnet (f.eks. leksikalsk analyse) for en stund når den har brug for en del af dens uddata.

Vi viser her en implementation af Datalog — leksikalsk analyse og parsing kombineret med en fortolker — hvor de to førstenævnte samarbejder som antydet om at opbygge en repræsentation af det aktuelle Datalog program som et abstrakte syntakstræ, som derefter overgives helt til fortolkningen. Havde vi haft at gøre med et oversætter i stedet for en fortolker, ville det have været naturligt ydermere at lave et samarbejde som antydet også med oversættelsesfase, så den ville være den overordnede proces, som rekvirerede stumper af syntakstræer, når det var nødvendigt — og hvor man iøvrigt undgik eksplicit konstruktion af syntakstræer, når det var muligt (sådan som vi har diskuteret det tidligere).

Vi beskriver her kort strukturen i fortolkeren, algoritmen samt ekstra datastrukturer. Den samlede programtekst er gengivet i appendices A og B i henholdvis Simula og Java; bemærk at navne på klasser, procedurer osv. af praktiske grunde er skrevet på engelsk og altså ikke nødvendigvis svarer til de navne, vi har brugt i de tidligere afsnit.

14.1 En abstrakt fortolkeralgoritme for Datalog

Strukturen af vor fortolker er inspireret af (Kahn, 1984), som benytter en repræsentation af variabelbindinger baseret på strukturdeling (Boyer, Moore, 1972), som har været væsentlig for at Prolog har kunnet udvikle sig til et praktisk anvendeligt og effektivt implementeret sprog.

De bånd på variablene, som bliver registreret under unification af et aktuelt mål med hovedet af en regel, kan vi repræsentere eksplicit i fortolkeren. Når en variabel `X` i princippet skal erstattes med et andet argument, `A`, udi i listen af mål, som venter på at bliver udført, noteres blot en *binding*.

```
X → A
```

Disse variabelbindinger er mere generelle end det, vi kender fra imperative og objektorienterede programmeringssprog, da variable kan bindes, ikke blot til konkrete værdier, men også til andre variable.

Disse bindinger findes ved at sammenholde det aktuelle mål med hovedet af en udvalgt klausul. Betragt som et eksempel følgende mål.

```
umage(A, B)
```

Sammenholdes dette mål med hovedet af reglen

```
umage(X, Y):- stor(X), lille(Y).
```

skabes bindingerne

```
A → X
B → Y.
```

Sammenholdes nu første mål i kroppen,

```
stor(X),
```

med faktummet

```
stor(elefant),
```

så skabes bindingen

```
X → elefant.
```

På tilsvarende måde undersøges

```
lille(Y)
```

og vi ender op med følgende, totale sæt af bindinger:

```
A → X
B → Y
X → elefant
Y → kylling
```

Altså er det vist, at værdierne `elefant` og `kylling` for de respektive variable `A` og `B` (via `X` og `Y`) gør forespørgslen

```
umage(A, B)
```

sand.

Resultatet af at sammenholde et aktuelt mål med hovedet af en regel kan enten resultere i skabelse af nye bindinger — eller i en konstatering af, at den givne regel ikke kan bruges. Reglen

```
umage(X, Y):- stor(X), lille(Y).
```

er eksempelvis ikke velegnet til at vise

```
stegt(kylling).
```

Unification kan beskrives vha. følgende sekventielle algoritme. Den tager som input to mål, et sæt eksisterende bindinger og producerer enten et udvidet sæt bindinger eller den specielle værdi *Fail*. Algoritmen benytter sig af en hjælpeprocedure ved navn »dereference«.

Algoritme 14.1 Unification.

Unify $\mathcal{M}_1$ og $\mathcal{M}_2$ givet bindingssæt $\mathcal{B}$:

- hvis $\mathcal{M}_1$ og $\mathcal{M}_2$ har forskellige prædikatsymboler
  eller forskelligt antal argumenter, så
    afslut med *Fail*,
- ellers lad
    $\mathcal{M}_1 = \mathrm{p}(\mathrm{p}_{1,1}, \ldots, \mathrm{p}_{1,\mathrm{n}})$,
    $\mathcal{M}_2 = \mathrm{p}(\mathrm{p}_{2,1}, \ldots, \mathrm{p}_{2,\mathrm{n}})$ og
    $\mathcal{B}' = \mathcal{B}$,
- for $i = 1, \ldots, n$ udfør
  - $\mathrm{p}_1 :=$ dereference$(\mathrm{p}_{1,i}, \mathcal{B}')$
  - $\mathrm{p}_2 :=$ dereference$(\mathrm{p}_{2,i}, \mathcal{B}')$
  - hvis p_1 er en variabel, så
      $\mathcal{B}' := \mathcal{B}' + [\mathrm{p}_1 \rightarrow \mathrm{p}_2]$,
  - ellers hvis p_2 er en variabel, så
      $\mathcal{B}' := \mathcal{B}' + [\mathrm{p}_2 \rightarrow \mathrm{p}_1]$,
  - ellers hvis p_1 og p_2 begge er atomer, så
      hvis $\mathrm{p}_1 \neq \mathrm{p}_2$ så afslut med *Fail*,
- afslut med $\mathcal{B}'$

Dereference(*arg*, $\mathcal{B}$):
- hvis *arg* er et atom, så afslut med *arg*,
- hvis *arg* er en variabel, så
  - hvis *arg* ikke er bundet i $\mathcal{B}$, så afslut med *arg*,
  - ellers er *arg* bundet til *arg*′ i $\mathcal{B}$,
    afslut med dereference(*arg*′, $\mathcal{B}$)

Der resterer blot en enkelt, lille præcisering i algoritmen, idet dannelse af ny binding skal undertrykkes i det tilfælde at p_1 og p_2 er én og samme variabel. Ellers, hvis vi f.eks. foretog handlingen $\mathcal{B}' := \mathcal{B}' + [\mathtt{X} \rightarrow \mathtt{X}]$, ville efterfølgende kald af Dereference$(\mathtt{X}, \mathcal{B}')$ gå i uendelig løkke.

Ved hjælp af denne formulering af unification kan vi præsentere en effektiv fortolker for Datalog — som er velegnet til at implementere på en konventionel datamaskine. Denne algoritme tager nu som input den liste af mål,

som skal udføres, samt et sæt bindinger; det »yderste« kald forventes foretaget med en forespørgsel og et tomt sæt bindinger.

Algoritme 14.2 Sekventiel udførelse af generelle forespørgsler i et program $\mathcal{P}$; unification implementeret ved bindingssæt.

At udføre en liste af mål, $\mathcal{L}$, under et sæt bindinger, $\mathcal{B}$:

- hvis $\mathcal{L}$ er tom, hejs et flag, $\boxed{Q.E.D.}$, og afslut,
- ellers lad $\mathcal{L} = \mathcal{M}_1, \ldots, \mathcal{M}_m$
- for alle klausuler, $\mathcal{K}$ i $\mathcal{P}$, udfør
  - dan en variant af $\mathcal{K}$, $\mathcal{K}'$, ved at ombenævne variable, så de ikke falder sammen med variable i $\mathcal{L}$,
    $\mathcal{K}' = \mathcal{N}_0 :- \mathcal{N}_1, \ldots, \mathcal{N}_n$
  - kald Unify $\mathcal{M}_1$ og $\mathcal{N}_0$ givet $\mathcal{B}$ og benævn resultatet $\mathcal{B}'$,
  - hvis $\mathcal{B}' \neq \mathit{Fail}$, så
    udfør $\mathcal{N}_1, \ldots, \mathcal{N}_n, \mathcal{M}_2, \ldots, \mathcal{M}_m$ under $\mathcal{B}'$

Bindingssættene vil vokse, når der foretages unification, og skrumpe ind, når der backtrackes. Én gang oprettede bindinger bliver aldrig ændret, højst smidt væk. Bindingssæt kan altså implementeres som en stak. Det vil i algoritmen ovenfor sige, at det at gå fra en udgave af $\mathcal{B}'$ tilbage til den oprindelige $\mathcal{B}$, er en simpel afstakningsoperation.

Grunden til, vi ombenævner variable, er for at undgå utilsigtet sammenblanding af variable i forskellige klausuler og forskellige instanser heraf. I praksis kan man foretage ombenævnelse ved at forsyne alle variable med et niveaunummer, som forøges for hvert rekursivt kald af fortolkeren. Udvælges for eksempel klausulen

```
umage(X, Y):- lille(X), stor(Y).
```

på niveau 27, benyttes varianten

```
umage(X27, Y27):- lille(X27), stor(Y27).
```

Alle variable i den liste af mål, der er ved at blive undersøgt, vil have niveaunumre på 26 eller derunder.

I stedet for at fortolkeren blot hejser et flag, når den har fundet en løsning, kunne den her udskrive den løsning den har fundet. Og den løsning,

brugeren er interesseret i, er netop værdierne bundet til variablene på yderste niveau, f.eks. med niveaunummer 0. (Naturligvis, de »derefence«-ede værdier — overvej!). Her kan man yderligere lade fortolkeren spørge brugeren, hvorvidt der ønskes alternative løsninger til den allerede fundne. Se proceduren `print_out_solution` i appendiks, som fremprovokerer alternative løsninger ved at simulere *Fail*.

14.2 Implementation i et objektorienteret sprog

14.2.1 Bindingssæt

Den centrale datastruktur i fortolkeren er repræsentationen af sæt af bindinger af logiske variable. I de abstrakte algoritmer benyttede vi dels at kunne slå variables værdier op i et givet bindingssæt, »dereference«, og dels at kunne udvide eksisterende bindingssæt. Hvis $\mathcal{B}$ et et sæt bindinger, benyttedes notationen

$$\mathcal{B} + [\mathtt{X} \rightarrow \mathtt{arg}]$$

til at betegne det bindingssæt, som indeholder de samme bindinger som $\mathcal{B}$ samt en binding af variablen `X` til argumentet `arg`. Derudover benyttede vi en speciel værdi, *Fail*, som det er praktisk at opfatte som en speciel slags bindingssæt. (*Fail* er det sæt bindinger, i hvilket intet udsagn er sandt).

For at kunne realisere de abstrakte algoritmer i Simula, er det nødvendigt at finde

- en datastruktur for bindingssæt,
- en implementation af »dereference«,
- en implementation af $\mathcal{B}+[\mathtt{X}\rightarrow\mathtt{arg}]$

Vi har valgt at repræsentere bindingssæt vha. en listestruktur, således at tilføjelse af en ny binding består i at hægte et nyt element foran på listen. De forskellige arter af bindingssæt er organiseret i et klassehierarki som følger.

```
class BINDING_SET;
  ! Bindings of Datalog variables;
begin
end BINDING_SET;


BINDING_SET class OK_BINDING_SET(var, val, rest);
  ref(VARIABLE) var;
```

```
  ref(ARGUMENT) val;
  ref(OK_BINDING_SET) rest;
begin
end OK_BINDING_SET;


OK_BINDING_SET class EMPTY_BINDING_SET;
  ! An empty or initial binding set;
begin
  ! Attributes are ignored;
end EMPTY_BINDING_SET;


BINDING_SET class FAIL;
  ! The result of an unsuccesful matching;
begin
end FAIL;
```

Givet et sæt bindinger, `B`, kan et udvidet bindingssæt, `B1`, konstrueres som følger, idet `X` og `p` forventes at være passende referencer.

```
B1:- new OK_BINDING_SET(X, p, B)
```

Implementation af »dereference« er nu blot et spørgsmål om en simpel omskrivning af den abstrakte algoritme.

Proceduren, der varetager unification, `unify_goals`, er også en omskrivning af den tilsvarende, abstrakte algoritme; der er blot benyttet en anden kontrolstruktur til at gennemløbe listen af argumenter og til at teste, at antallet af argumenter i de respekte mål er ens. Af overskuelighedshensyn er en del af koden trukket ud i en hjælpe-procedure, `unify_arguments`.

For at forenkle den del af programmet, som tager sig af fortolkning, er variablenes niveaunumre placeret som attributter i de klasser, som retteligt beskriver Datalogs abstrakte syntaks. Det, at skabe varianter af klausuler, i hvilke variablene forsynes med et givet niveaunummer, varetages af den virtuelle procedure `new_copy`, som ligeledes er flettet ind i de syntaktiske klasser.

14.2.2 Fortolkeralgoritmen

Den abstrakte algoritme 14.2 er omskrevet stort set ordret til proceduren `prove`. Den eneste forskel er, at proceduren har to yderligere parametre, et niveaunummer og en »`exit_label`«. Niveaunummeret et et heltal, som forøges med 1 for hvert rekursive kald af fortolkeren. Som tidligere beskrevet benyttes niveaunummeret til at generere varianter af klausuler med garanteret »helt nye« variable. Afbrydelse af fortolkerprocessen i det tilfælde, brugeren ikke ønsker

at se alle løsninger, foretages ved at hoppe til `exit_label`, hvilket besørges af proceduren `print_out_solution`.

Endelig er algoritme 14.2's »for«-løkke, som itererer over programmets klausuler, implementeret ved en »while«-konstruktion, idet programmer er repræsenteret som lister af klausuler.

Proceduren `execute` er et »brugerhåndtag« til fortolkeren, som kaldes med en forespørgsel og et Datalogprogram. Dette procedure sætter så den rette starttilstand op for `prove`.

Appendiks A – Et Datalogsystem i Simula

Det Simulaprogram, som udgør Datalogsystemet, er her gengivet i sin fulde tekst. For ikke at komplicere Simulateksten mere end højst nødvendigt, er komplicerede skærmdialoger undgået. Datalogsystemet læser således altid et Datalogprogram fra en tekstfil med navnet

```
testprogram
```

og en (og kun en) forespørgsel fra en tekstfil med navn

```
testquery.
```

Det indlæste program og den indlæste forespørgsel udskrives på skærmen, vha. den tidligere beskrevne pretty-print-procedure. Såfremt der konstateres en syntaksfejl i Datalogteksten, udskrives kun en ikke særlig informativ fejlmeddelse.

```
begin

!****************************************
    S y n t a x   o f   D a t a l o g
 ****************************************;


  class CATEGORY;

    ! The class of syntactic objects in
      the Datalog programming language;
    virtual: procedure read, pretty_print;
             ref(CATEGORY) procedure new_copy;
             ! the latter used by the interpreter;
  begin
  end CATEGORY;
```

```
CATEGORY class PROGRAM;

  ! Datalog programs;

begin
  ! A list of CLAUSEs;
  ref(CLAUSE) first;
  ref(PROGRAM) rest;

  procedure read(source);
    ref(LEXICAL_SCANNER) source;
  begin
    first:- new CLAUSE; first.read(source);
    if not source.look_ahead is END_OF_FILE then
      begin rest:- new PROGRAM; rest.read(source) end
  end read;

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

end PROGRAM;

CATEGORY class CLAUSE;

  ! Datalog clauses;

begin
  ref(GOAL) head;
  ref(GOAL_LIST) body; ! Empty body signifies a fact;

  procedure read(source);
    ref(LEXICAL_SCANNER) source;
  begin
    head:- new GOAL; head.read(source);
    inspect source.look_ahead
      when COLON_DASH do
        begin
          source.read_symbol;
          body:- new GOAL_LIST;
```

```
          body.read(source)
        end
      when PERIOD do
    otherwise error(":- or . expected");
    if source.look_ahead is PERIOD then
      source.read_symbol
    else
      error(". expected")
  end read;

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

  ! The following used by the interpreter;

  ref(CLAUSE) procedure new_copy(level);
    integer level;
  begin
    ref(CLAUSE) cl; cl:- new CLAUSE;
    cl.head:- head.new_copy(level);
    cl.body:- if body == none then none
              else body.new_copy(level);
    new_copy:- cl
  end new_copy;

end CLAUSE;

CATEGORY class GOAL;

  ! Datalog goals;

begin
  ref(ATOM) predicate;
  ref(ARGUMENT_LIST) arguments;

  procedure read(source);
    ref(LEXICAL_SCANNER) source;
  begin
```

```
    predicate:- new ATOM;
    predicate.read(source);
    if source.look_ahead is BEGIN_PAR then
      source.read_symbol
    else error("( expected");
    arguments:- new ARGUMENT_LIST;
    arguments.read(source);
    if source.look_ahead is END_PAR then
      source.read_symbol
    else error(") expected")
  end read;

  procedure pretty_print;
  begin
    predicate.pretty_print;
    outtext("(");
    arguments.pretty_print;
    outtext(")")
  end pretty_print;

  ! The following used by the interpreter;

  ref(GOAL) procedure new_copy(level);
    integer level;
  begin
    ref(GOAL) g; g:- new GOAL;
    g.predicate:- predicate.new_copy(level);
    g.arguments:- arguments.new_copy(level);
    new_copy:- g
  end new_copy;

end GOAL;

CATEGORY class GOAL_LIST;

begin
  ref(GOAL) first;
  ref(GOAL_LIST) rest;

  procedure read(source);
    ref(LEXICAL_SCANNER) source;
  begin
    first:- new GOAL;
    first.read(source);
```

```
    if source.look_ahead is COMMA then
      begin
        source.read_symbol;
        rest:- new GOAL_LIST;
        rest.read(source)
      end
  end read;

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

  ! The following used by the interpreter;

  ref(GOAL_LIST) procedure new_copy(level);
    integer level;
  begin
    ref(GOAL_LIST) gl; gl:- new GOAL_LIST;
    gl.first:- first.new_copy(level);
    gl.rest:- if rest == none then none
              else rest.new_copy(level);
    new_copy:- gl
  end new_copy;

end GOAL_LIST;

CATEGORY class ARGUMENT;

  ! Arguments in Datalog goals;

  ! The following used by the interpreter;
  virtual: boolean procedure equal;
begin
  ! An ATOM or a VARIABLE;
end ARGUMENT;
```

```
CATEGORY class ARGUMENT_LIST;

begin
  ref(ARGUMENT) first;
  ref(ARGUMENT_LIST) rest;

  procedure read(source);
    ref(LEXICAL_SCANNER) source;
  begin
    inspect source.look_ahead
      when ATOM_LEX do
        begin
          first:- new ATOM;
          first.read(source)
        end
      when VARIABLE_LEX do
        begin
          first:- new VARIABLE;
          first.read(source)
        end
    otherwise error("argument expected");
    if source.look_ahead is COMMA then
      begin
        source.read_symbol;
        rest:- new ARGUMENT_LIST;
        rest.read(source)
      end
  end read;

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

  ! The following used by the interpreter;

  ref(ARGUMENT_LIST) procedure new_copy(level);
    integer level;
  begin
    ref(ARGUMENT_LIST) al; al:- new ARGUMENT_LIST;
```

```
    al.first:- first.new_copy(level);
    al.rest:- if rest == none then none
              else rest.new_copy(level);
    new_copy:- al
  end new_copy;

end ARGUMENT_LIST;

ARGUMENT class ATOM;

  ! Atomic constant in Datalog;

begin
  text id;

  procedure read(source);
    ref(LEXICAL_SCANNER) source;
  begin
    if source.look_ahead is ATOM_LEX then
      id:- source.read_symbol qua ATOM_LEX.lex
    else
      error("atom expected")
  end read;

  procedure pretty_print;
    outtext(id);

  ! The following used by the interpreter;

  boolean procedure equal(arg);
    ref(ARGUMENT) arg;
  equal:= arg is ATOM and then id = arg qua ATOM.id;

  ref(ATOM) procedure new_copy(level);
    integer level;
  begin
    ref(ATOM) atm; atm:- new ATOM;
    atm.id:- id;
    new_copy:- atm
  end new_copy;

end ATOM;
```

```
ARGUMENT class VARIABLE;

  ! Variables in Datalog;

begin
  text id;

  procedure read(source);
    ref(LEXICAL_SCANNER) source;
  begin
    if source.look_ahead is VARIABLE_LEX then
      id:- source.read_symbol qua VARIABLE_LEX.lex
    else
      error("variable expected")
  end read;

  procedure pretty_print;
  begin
    outtext(id);
    if level <> 0 then
      begin
        outtext("/");
        outint(level,0)
      end
  end pretty_print;

  ! The following used by the interpreter;

  integer level;

  boolean procedure equal(arg);
    ref(ARGUMENT) arg;
  equal:= arg is VARIABLE
            and then id = arg qua VARIABLE.id
            and then level = arg qua VARIABLE.level;

  ref(VARIABLE) procedure new_copy(level);
    integer level;
  begin
    ref(VARIABLE) v; v:- new VARIABLE;
    v.level:= level;
    v.id:- id;
    new_copy:- v
```

```
    end new_copy;

  end VARIABLE;

  CATEGORY class QUERY;

    ! Query in Datalog;

  begin
    ref(GOAL_LIST) goals;

    procedure read(source);
      ref(LEXICAL_SCANNER) source;
    begin
      goals:- new GOAL_LIST;
      goals.read(source);
      if not source.look_ahead is PERIOD then
        error(". expected")
      else source.read_symbol
    end read;

    procedure pretty_print;
    begin
      goals.pretty_print;
      outtext(".");
      outimage
    end pretty_print;

  end QUERY;

!************************************
   L e x i c a l   s y n t a x   f o r
          D a t a l o g
 ************************************;

  class LEXICAL_SYMBOL;
    ! The smallest syntactically and semantically
      significant objects;
  begin
  end LEXICAL_SYMBOL;

  LEXICAL_SYMBOL class ATOM_LEX(lex);
    text lex;
```

```
    ! The lexical symbol denoting an ATOM;
  begin
  end ATOM_LEX;

  LEXICAL_SYMBOL class VARIABLE_LEX(lex);
    text lex;
    ! The lexical symbol denoting a VARIABLE;
  begin
  end VARIABLE_LEX;

  LEXICAL_SYMBOL class COLON_DASH;
    ! The symbol ":-" ;
  begin
  end COLON_DASH;

  LEXICAL_SYMBOL class COMMA;
    ! The symbol "," ;
  begin
  end COMMA;

  LEXICAL_SYMBOL class PERIOD;
    ! The symbol "." ;
  begin
  end PERIOD;

  LEXICAL_SYMBOL class BEGIN_PAR;
    ! The symbol "(" ;
  begin
  end BEGIN_PAR;

  LEXICAL_SYMBOL class END_PAR;
    ! The symbol ")" ;
  begin
  end ENDPAR;

  LEXICAL_SYMBOL class END_OF_FILE;
    ! Signals the end of a Datalog program file;
  begin
  end END_OF_FILE;

!****************************************
   T h e   l e x i c a l   s c a n n e r
 ****************************************;
```

```
class LEXICAL_SCANNER(file_name);
  value file_name; text file_name;
begin
  ref(LEXICAL_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(LEXICAL_SYMBOL) procedure read_symbol;
  begin
    read_symbol:- look_ahead;
    ! set look_ahead to the next LEXICAL_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 <= 'z' 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 <= 'Z' then
      begin
        text string; string:- blanks(100);
        while letter(character_look_ahead) do
          string.putchar(character_read);
        look_ahead:- new VARIABLE_LEX(string.strip)
      end
    else if character_look_ahead = ',' then
      begin
        look_ahead:- new COMMA;
        character_read
      end
    else if character_look_ahead = '.' then
      begin
```

```
          look_ahead:- new PERIOD;
          character_read
        end
      else if character_look_ahead = '(' then
        begin
          look_ahead:- new BEGIN_PAR;
          character_read
        end
      else if character_look_ahead = ')' then
        begin
          look_ahead:- new END_PAR;
          character_read
        end
      else if character_look_ahead = ':' then
        begin
          character_read;
          if character_look_ahead = '-' then
            begin
              character_read;
              look_ahead:- new COLON_DASH
            end
          else
            error("- expected after :")
        end
      else error("illegal character")
    end read_symbol;

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

    ! Initialize look_ahead:  ;
    character_read;
    read_symbol
  end LEXICAL_SCANNER;

!*******************************
   T h e   i n t e r p r e t e r
 *******************************;

  class BINDING_SET;
    ! Bindings of Datalog variables;
  begin
  end BINDING_SET;
```

```
BINDING_SET class OK_BINDING_SET(var, val, rest);
  ref(VARIABLE) var;
  ref(ARGUMENT) val;
  ref(OK_BINDING_SET) rest;
begin
end OK_BINDING_SET;


OK_BINDING_SET class EMPTY_BINDING_SET;
  ! An empty or initial binding set;
begin
  ! Attributes are ignored;
end EMPTY_BINDING_SET;


BINDING_SET class FAIL;
  ! The result of an unsuccesful matching;
begin
end FAIL;


ref(ARGUMENT) procedure dereference(arg, bindings);
  ref(ARGUMENT) arg;
  ref(OK_BINDING_SET) bindings;
  ! returns the deepest value of arg;
inspect arg
  when ATOM do dereference:- arg
  when VARIABLE do
    begin
      ref(OK_BINDING_SET) rest_bindings;
      rest_bindings:- bindings;
      while not rest_bindings is EMPTY_BINDING_SET do
        if rest_bindings.var.equal(arg) then
          begin
            dereference:-
              dereference(rest_bindings.val, bindings);
            go to exit_dereference
          end
        else rest_bindings:- rest_bindings.rest;
      dereference:- arg; ! uninstantiated variable;
exit_dereference:
    end;


ref(BINDING_SET) procedure bind(old_bindings,
                                the_var, the_val);
  ref(OK_BINDING_SET) old_bindings;
  ref(VARIABLE) the_var;
  ref(ARGUMENT) the_val;
  ! produces an extension of this BINDING_SET,
```

```
    the_var is expected not to be previously bound
    and the_val is expected to be an atom or
    an unbound variable;
begin
  if the_val is VARIABLE
     and then the_val qua VARIABLE.equal(the_var) then
    bind:- old_bindings
  else
    bind:- new OK_BINDING_SET(the_var, the_val,
                              old_bindings)
end bind;

ref(BINDING_SET) procedure unify_goals(goal1, goal2,
                                       old_bindings);
  ref(GOAL) goal1, goal2;
  ref(OK_BINDING_SET) old_bindings;
  ! Match goal1 and goal2 in old_bindings,
    producing an extended OK_BINDING_SET or FAIL;
begin
  ref(ARGUMENT_LIST) arg_list1, arg_list2;
  ref(BINDING_SET) current_bindings;
  current_bindings:-
     if goal1.predicate.equal(goal2.predicate) then
       old_bindings
     else new FAIL;
  arg_list1:- goal1.arguments;
  arg_list2:- goal2.arguments;
  while arg_list1 =/= none and arg_list2 =/= none
        and not current_bindings is FAIL do
    begin
      current_bindings:-
          unify_arguments(arg_list1.first,
                          arg_list2.first,
                  current_bindings qua OK_BINDING_SET);
      arg_list1:- arg_list1.rest;
      arg_list2:- arg_list2.rest
    end;
  unify_goals:-
      if arg_list1 =/= none
         or arg_list2 =/= none then
        new FAIL
      else current_bindings;
end unify_goals;

ref(BINDING_SET) procedure
    unify_arguments(arg1, arg2, old_bindings);
  ref(ARGUMENT) arg1, arg2;
```

```
  ref(OK_BINDING_SET) old_bindings;
begin
  arg1:- dereference(arg1, old_bindings);
  arg2:- dereference(arg2, old_bindings);
  if arg1 is ATOM and arg2 is ATOM then
    unify_arguments:- if arg1.equal(arg2) then
                        old_bindings
                      else new FAIL
  else if arg1 is VARIABLE then
    unify_arguments:- bind(old_bindings, arg1, arg2)
  else
    unify_arguments:- bind(old_bindings, arg2, arg1)
end unify_arguments;


procedure print_out_solution(bindings, exit_label);
  ref(OK_BINDING_SET) bindings;
  label exit_label;
begin
  character answer;
  ref(OK_BINDING_SET) rest_bindings;
  rest_bindings:- bindings;
  while not rest_bindings is EMPTY_BINDING_SET do
    begin
      if rest_bindings.var.level = 0 then
        begin
          outtext("    ");
          rest_bindings.var.pretty_print;
          outtext(" = ");
          dereference(rest_bindings.val, bindings)
              .pretty_print; outimage
        end;
      rest_bindings:- rest_bindings.rest
    end;
  outtext("yes"); outimage;
try_again:
  outtext("more solutions! (y/n)"); outimage;
  if not lastitem then
    answer:= inchar;
  if answer='y' then
    !proceed;
  else if answer='n' then
    go to exit_label
  else
    go to try_again
end print_out_solution;


procedure prove(list_of_goals, bindings, prog,
```

```
                level, exit_label);
    ref(OK_BINDING_SET) bindings;
    ref(GOAL_LIST) list_of_goals;
    ref(PROGRAM) prog;
    integer level;
    label exit_label;
  begin
    ref(PROGRAM) rules_left;
    if list_of_goals == none then
      begin
        print_out_solution(bindings, exit_label);
        go to exit_prove
      end;
    rules_left:- prog;
    while rules_left =/= none do
      begin
        ref(BINDING_SET) bindings1; ref(CLAUSE) rule;
        rule:- rules_left.first.new_copy(level);
        rules_left:- rules_left.rest;
        bindings1:-
            unify_goals(list_of_goals.first, rule.head,
                        bindings);
        inspect bindings1 when OK_BINDING_SET do
          prove(append_goals(rule.body,
                             list_of_goals.rest),
                bindings1, prog, level + 1, exit_label)
      end while;
  exit_prove:
  end prove;

  procedure execute(question, prog);
    ref(QUERY) question; ref(PROGRAM) prog;
    ! The top-level prove procedure;
  begin
    ref(OK_BINDING_SET) initial_state;
    initial_state:-
        new EMPTY_BINDING_SET(none, none, none);
    prove(question.goals, initial_state, prog, 1,
          exit_execute);
  exit_execute:
    outtext("no (more) solutions"); outimage
  end execute;

!***** Auxiliary procedure used in the interpreter: ;

  ref(GOAL_LIST) procedure append_goals(goals1, goals2);
```

```
    ref(GOAL_LIST) goals1, goals2;
  if goals1 == none then
    append_goals:- goals2
  else begin
         ref(GOAL_LIST) result;
         result:- new GOAL_LIST;
         result.first:- goals1.first;
         result.rest:- append_goals(goals1.rest, goals2);
         append_goals:- result
       end;


!****************
   Main program
 ****************;


  ref(LEXICAL_SCANNER) source;
  ref(PROGRAM) datalog_program;
  ref(QUERY) question;


  outimage;
  outtext("Reading program from file TESTPROGRAM");
  source:- new LEXICAL_SCANNER("testprogram");
  datalog_program:- new PROGRAM;
  datalog_program.read(source);
  datalog_program.pretty_print;


  outimage; outimage;
  outtext("Reading program from file TESTQUERY");
  source:- new LEXICAL_SCANNER("testquery");
  question:- new QUERY;
  question.read(source);
  question.pretty_print;


  outimage;
  outtext("Executing..."); outimage;


  execute(question, datalog_program)

end  Datalog interpreter;
```

Appendiks B
– Et Datalogsystem i Java

Her følger uden yderligere kommentarer Simula-programmet fra Appendix A omskrevet systematisk til Java.

```
import java.io.*;
import com.mw.SystemInput;

//*******************************************
//   S y n t a x   o f   D a t a l o g
// *******************************************;

abstract class CATEGORY {
    // The class of syntactic objects in
    // the Datalog programming language
   abstract void read(LEXICAL_SCANNER source);
   abstract void pretty_print();
   abstract CATEGORY new_copy(int level);
    //the latter used by the interpreter;
}

class PROGRAM extends CATEGORY {
  // Datalog programs

    // A list of CLAUSEs
    CLAUSE first;
    PROGRAM rest;

    void read(LEXICAL_SCANNER source) {
    first = new CLAUSE(); first.read(source);
       if (!(source.look_ahead instanceof END_OF_FILE))
      { rest = new PROGRAM(); rest.read(source); }
    }

    void pretty_print() {
    first.pretty_print();
```

```
        if (rest != null)
          rest.pretty_print();
     }

    CATEGORY new_copy(int level) { return null; }

}

class CLAUSE extends CATEGORY {

// Datalog clauses;

     GOAL head;
     GOAL_LIST body; // Empty body signifies a fact

     void read(LEXICAL_SCANNER source) {
        head = new GOAL(); head.read(source);
       if (source.look_ahead instanceof COLON_DASH) {
             source.read_symbol();
             body = new GOAL_LIST();
             body.read(source);
         }
         else if (!(source.look_ahead instanceof PERIOD))
        throw new RuntimeException(":- or . expected");
        if (source.look_ahead instanceof PERIOD)
         source.read_symbol();
         else
        throw new RuntimeException(":- or . expected");
     }

     void pretty_print() {
      System.out.println();
        head.pretty_print();
        if (body != null)
          { System.out.print(":-"); body.pretty_print(); }
        System.out.print(".");
    }

     // The following used by the interpreter

    CATEGORY new_copy(int level) {
      CLAUSE cl = new CLAUSE();
        cl.head = (GOAL) head.new_copy(level);
        cl.body = body == null ? null :
                             (GOAL_LIST) body.new_copy(level);
        return cl;
     }
```

```
}


class GOAL extends CATEGORY {

    // Datalog goals

    ATOM predicate;
    ARGUMENT_LIST parameters;

    void read(LEXICAL_SCANNER source) {
       predicate = new ATOM();
       predicate.read(source);
       if (source.look_ahead instanceof BEGIN_PAR)
         source.read_symbol();
       else
       throw new RuntimeException("( expected");
       parameters = new ARGUMENT_LIST();
       parameters.read(source);
       if (source.look_ahead instanceof END_PAR)
         source.read_symbol();
       else
       throw new RuntimeException(") expected");
    }

    void pretty_print() {
       predicate.pretty_print();
       System.out.print("(");
       parameters.pretty_print();
       System.out.print(")");
    }

    // The following used by the interpreter

    CATEGORY new_copy(int level) {
       GOAL g = new GOAL();
       g.predicate = (ATOM) predicate.new_copy(level);
       g.parameters =
             (ARGUMENT_LIST) parameters.new_copy(level);
      return g;
    }

}


class GOAL_LIST extends CATEGORY {
```

```
    GOAL first;
    GOAL_LIST rest;

    void read(LEXICAL_SCANNER source) {
       first = new GOAL();
       first.read(source);
       if (source.look_ahead instanceof COMMA) {
           source.read_symbol();
           rest = new GOAL_LIST();
           rest.read(source);
        }
    }

    void pretty_print() {
     System.out.println();
     System.out.print("    ");
       first.pretty_print();
       if (rest != null) {
           System.out.print(",");
           rest.pretty_print();
        }
   }

   // The following used by the interpreter

    CATEGORY new_copy(int level) {
       GOAL_LIST gl = new GOAL_LIST();
       gl.first = (GOAL) first.new_copy(level);
       gl.rest = rest == null ? null :
                      (GOAL_LIST) rest.new_copy(level);
      return gl;
    }

}


abstract class ARGUMENT extends CATEGORY {

   // Arguments in Datalog goals

    // The following used by the interpreter
    abstract boolean equal(ARGUMENT arg);

    // An ATOM or a VARIABLE
}
```

```
class ARGUMENT_LIST extends  CATEGORY {

   ARGUMENT first;
    ARGUMENT_LIST rest;

    void read(LEXICAL_SCANNER source) {
     if (source.look_ahead instanceof ATOM_LEX) {
            first = new ATOM();
            first.read(source);
         }
         else if (source.look_ahead instanceof VARIABLE_LEX) {
            first = new VARIABLE();
            first.read(source);
       }
       else
       throw new RuntimeException("argument expected");
       if (source.look_ahead instanceof COMMA) {
           source.read_symbol();
           rest = new ARGUMENT_LIST();
           rest.read(source);
       }
   }

    void pretty_print() {
       first.pretty_print();
       if (rest != null) {
           System.out.print(", ");
           rest.pretty_print();
        }
    }

    // The following used by the interpreter

    CATEGORY new_copy(int level) {
       ARGUMENT_LIST al = new ARGUMENT_LIST();
       al.first = (ARGUMENT) first.new_copy(level);
       al.rest = rest == null ? null :
                    (ARGUMENT_LIST) rest.new_copy(level);
       return al;
    }

}


class ATOM extends ARGUMENT {;
```

```
  // Atomic constant in Datalog

    String id;

    void read(LEXICAL_SCANNER source) {
       if (source.look_ahead instanceof ATOM_LEX)
         id = ((ATOM_LEX) source.read_symbol()).lex;
       else
       throw new RuntimeException("atom expected");
    }

    void pretty_print() {
       System.out.print(id);
    }

    // The following used by the interpreter

    boolean equal(ARGUMENT arg) {
     return arg instanceof ATOM
               && id.equals(((ATOM) arg).id);
    }

    CATEGORY new_copy(int level) {
       ATOM atm = new ATOM();
       atm.id = id;
       return atm;
    }

}


class VARIABLE extends ARGUMENT {

   // Variables in Datalog

    String id;

    void read(LEXICAL_SCANNER source) {
     if (source.look_ahead instanceof VARIABLE_LEX)
         id = ((VARIABLE_LEX) source.read_symbol()).lex;
       else
       throw new RuntimeException("variable expected");
    }

    void pretty_print() {
      System.out.print(id);
       if (level != 0) {
```

```
            System.out.print("/");
            System.out.print(level);
         }
   }

    // The following used by the interpreter

    int level;

    boolean equal(ARGUMENT arg) {
    return arg instanceof VARIABLE
               && id.equals(((VARIABLE) arg).id)
               && level == ((VARIABLE) arg).level;
   }

    CATEGORY new_copy(int level) {
    VARIABLE v = new VARIABLE();
       v.level = level;
       v.id = id;
       return v;
    }

}


class QUERY extends CATEGORY {

    // Query in Datalog

    GOAL_LIST goals;

    void read(LEXICAL_SCANNER source) {
       goals = new GOAL_LIST();
       goals.read(source);
       if (!(source.look_ahead instanceof PERIOD))
       throw new RuntimeException(". expected");
       else source.read_symbol();
    }

    void pretty_print() {
       goals.pretty_print();
       System.out.println(".");
    }

CATEGORY new_copy(int level) { return null; }

}
```

```
//*********************************
//  Lexical syntax for Datalog
//*********************************


class LEXICAL_SYMBOL {
   // The smallest syntactically and semantically
    //  significant objects
}

class ATOM_LEX extends LEXICAL_SYMBOL {
    // The lexical symbol denoting an ATOM
String lex;

   ATOM_LEX(String lex) {this.lex = lex; }
}

class VARIABLE_LEX extends LEXICAL_SYMBOL {
    // The lexical symbol denoting a VARIABLE

    String lex;

   VARIABLE_LEX(String lex) {this.lex = lex; }
}

class COLON_DASH extends LEXICAL_SYMBOL {
    // The symbol ":-"
}

class COMMA extends LEXICAL_SYMBOL {
    // The symbol ","
}

class PERIOD extends LEXICAL_SYMBOL {
    // The symbol "."
}

class BEGIN_PAR extends LEXICAL_SYMBOL {
    // The symbol "("
}

class END_PAR extends LEXICAL_SYMBOL {
    // The symbol ")"
}
```

```
class END_OF_FILE extends LEXICAL_SYMBOL {
    // Signals the end of a Datalog program file
}


//****************************************
//  T h e   l e x i c a l   s c a n n e r
//****************************************


class LEXICAL_SCANNER {

    LEXICAL_SYMBOL look_ahead;
    char character_look_ahead;
    FileReader file;
    boolean endfile;

    LEXICAL_SCANNER(String file_name) {
     try {
     file = new FileReader(file_name);

     // Initialize look_ahead:
     character_read();
    read_symbol();
     }
     catch (Exception e) {
     throw new RuntimeException(
                 "File error: " + file_name);
     }
    }

    char character_read() {
       char character = character_look_ahead;
       try {
        character_look_ahead = (char) file.read();
        if (character_look_ahead == (char) -1)
           endfile = true;
        }
        catch (IOException e) { endfile = true; }
        return character;
    }

    LEXICAL_SYMBOL read_symbol()  {
       LEXICAL_SYMBOL symbol = look_ahead;
       // set look_ahead to the next LEXICAL_SYMBOL
       while (!endfile &&
               Character.isWhitespace(character_look_ahead))
```

```
            character_read();
          if (endfile) {
            look_ahead = new END_OF_FILE();
            return symbol;
           }
         if ('a' <= character_look_ahead &&
         character_look_ahead <= 'z') {
             StringBuffer string = new StringBuffer();
          while (Character.isLetter(character_look_ahead))
                string.append(character_read());
              look_ahead = new ATOM_LEX(new String(string));
          }
          else if ('A' <= character_look_ahead &&
                   character_look_ahead <= 'Z') {
           StringBuffer string = new StringBuffer();
          while (Character.isLetter(character_look_ahead))
                string.append(character_read());
              look_ahead = new VARIABLE_LEX(new String(string));
          }
          else if (character_look_ahead == ',') {
              look_ahead = new COMMA();
              character_read();
          }
          else if (character_look_ahead == '.') {
              look_ahead = new PERIOD();
              character_read();
           }
          else if (character_look_ahead == '(') {
              look_ahead = new BEGIN_PAR();
              character_read();
           }
          else if (character_look_ahead == ')') {
              look_ahead = new END_PAR();
              character_read();
           }
          else if (character_look_ahead == ':') {
              character_read();
              if (character_look_ahead == '-') {
                  character_read();
                  look_ahead = new COLON_DASH();
               }
            else
                throw new RuntimeException("- expected after :");
          }
          else throw new RuntimeException("illegal character");
          return symbol;
      }
```

```
 }

//*********************************
//  T h e   i n t e r p r e t e r
//*********************************

class BINDING_SET {
   // Bindings of Datalog variables
}

class OK_BINDING_SET extends BINDING_SET {
OK_BINDING_SET(VARIABLE var, ARGUMENT val,
               OK_BINDING_SET rest) {
this.var = var; this.val = val; this.rest = rest;
}

    VARIABLE var;
    ARGUMENT val;
    OK_BINDING_SET rest;
}

class EMPTY_BINDING_SET extends OK_BINDING_SET {
    // An empty or initial binding set
    // Attributes are ignored
    EMPTY_BINDING_SET(VARIABLE var, ARGUMENT val,
                      OK_BINDING_SET rest) {
     super(var, val, rest);
    }
}

class FAIL extends BINDING_SET {
   // The result of an unsuccesful matching
}

class Exit extends Exception {}

class Interpreter {
ARGUMENT dereference(ARGUMENT param,
                     OK_BINDING_SET bindings) {
    // returns the deepest value of param;
    if (param instanceof ATOM)
    return param;
    if (param instanceof VARIABLE) {
         OK_BINDING_SET rest_bindings = bindings;
         while (!(rest_bindings instanceof EMPTY_BINDING_SET))
           if(rest_bindings.var.equal(param))
```

```
            return dereference(rest_bindings.val, bindings);
          else rest_bindings = rest_bindings.rest;
         }
         return param; // uninstantiated
}

BINDING_SET bind(OK_BINDING_SET old_bindings,
                 VARIABLE the_var,
                 ARGUMENT the_val) {
 // produces an extension of this BINDING_SET,
  // the_var is expected not to be previously bound and
  // the_val is expected to be an atom or an unbound variable
   if (the_val instanceof VARIABLE &&
       ((VARIABLE) the_val).equal(the_var))
    return old_bindings;
   return new OK_BINDING_SET(the_var, the_val, old_bindings);
  }

 BINDING_SET unify_goals(GOAL goal1, GOAL goal2,
                     OK_BINDING_SET old_bindings) {
  // Match goal1 and goal2 in old_bindings,
  // producing an extended OK_BINDING_SET or FAIL
   ARGUMENT_LIST param_list1, param_list2;
   BINDING_SET current_bindings = null;
   current_bindings =
        goal1.predicate.equal(goal2.predicate) ?
             (BINDING_SET) old_bindings : new FAIL();
   param_list1 = goal1.parameters;
   param_list2 = goal2.parameters;
   while (param_list1 != null && param_list2 != null &&
             !(current_bindings instanceof FAIL)) {
       current_bindings =
           unify_arguments(param_list1.first,
                           param_list2.first,
                          (OK_BINDING_SET) current_bindings);
       param_list1 = param_list1.rest;
       param_list2 = param_list2.rest;
     }
   return param_list1 != null || param_list2 != null
                     ? new FAIL() : current_bindings;
 }

BINDING_SET unify_arguments(ARGUMENT arg1, ARGUMENT arg2,
                        OK_BINDING_SET old_bindings) {
   arg1 = dereference(arg1, old_bindings);
   arg2 = dereference(arg2, old_bindings);
   if (arg1 instanceof ATOM && arg2 instanceof ATOM)
```

```
    return arg1.equal(arg2) ?
      (BINDING_SET) old_bindings : new FAIL();
if (arg1 instanceof VARIABLE)
    return bind(old_bindings, (VARIABLE) arg1, arg2);
    return bind(old_bindings, (VARIABLE) arg2, arg1);
}

void print_out_solution(OK_BINDING_SET bindings)
                                      throws Exit {
  OK_BINDING_SET rest_bindings = bindings;
  while (!(rest_bindings instanceof EMPTY_BINDING_SET)) {
      if (rest_bindings.var.level == 0) {
         System.out.print("    ");
          rest_bindings.var.pretty_print();
          System.out.print(" = ");
          dereference(rest_bindings.val, bindings)
                                  .pretty_print();
          System.out.println();
      }
       rest_bindings = rest_bindings.rest;
 }
  System.out.println("yes");
  while (true) {
  char answer;
  DataInputStream input =
               new DataInputStream(System.in);
  System.out.println("more solutions? (y/n)");
  try {
  answer = input.readLine().charAt(0);
  }
  catch(IOException e) { throw new Exit(); }
  if (answer == 'y')
  return;
  if (answer == 'n')
  throw new Exit();
 }
 }

void prove(GOAL_LIST list_of_goals, OK_BINDING_SET bindings,
       PROGRAM prog, int level) throws Exit {
  PROGRAM rules_left;
  if (list_of_goals == null) {
      print_out_solution(bindings);
      return;
     }
 rules_left = prog;
  while (rules_left != null) {
```

```
          BINDING_SET bindings1;
          CLAUSE rule =
              (CLAUSE) rules_left.first.new_copy(level);
          rules_left = rules_left.rest;
          bindings1 = unify_goals(list_of_goals.first,
                                  rule.head, bindings);
          if (bindings1 instanceof OK_BINDING_SET)
           prove(append_goals(rule.body,
                              list_of_goals.rest),
                     (OK_BINDING_SET) bindings1,
                       prog, level + 1);
         }
     }

   void execute(QUERY question, PROGRAM prog) {
    // The top-level prove procedure;
     OK_BINDING_SET initial_state;
     initial_state = new EMPTY_BINDING_SET(null, null, null);
     try {
     prove(question.goals, initial_state, prog, 1);
     }
     catch (Exit e) {}
     System.out.println("no (more) solutions");
    }

// ***** Auxiliary method used in the interpreter:

   GOAL_LIST append_goals(GOAL_LIST goals1, GOAL_LIST goals2) {
   if (goals1 == null)
   return goals2;
   GOAL_LIST result = new GOAL_LIST();
       result.first = goals1.first;
        result.rest = append_goals(goals1.rest, goals2);
       return result;
   }

}


public class Datalog {

// ***************
    //   Main program
  // ***************

  public static void main(String[] args) {
LEXICAL_SCANNER source;
```

```
PROGRAM datalog_program;
QUERY question;

System.out.println();
System.out.print("Reading program from file TESTPROGRAM");
System.out.flush();
source = new LEXICAL_SCANNER("testprogram");
    datalog_program = new PROGRAM();
datalog_program.read(source);
datalog_program.pretty_print();

System.out.println(); System.out.println();
System.out.print("Reading program from file TESTQUERY");
System.out.flush();
source = new LEXICAL_SCANNER("testquery");
question = new QUERY();
question.read(source);
question.pretty_print();

System.out.println();
System.out.println("Executing...");

SystemInput sysin = new SystemInput();
new Interpreter().execute(question, datalog_program);
sysin.dispose();
}
}
```

Litteratur

Abelson, H. og Sussman, G.J., *Structure and Interpretation of Computer Programs.* MIT Press & McGraw-Hill, 1985.

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

Aho, A.V, Ullman, J.D., *The theory of parsing, translation, and compiling. Volume I: Parsing.* Prentice-Hall 1972.

Barklund, J., Costantini, S., Dell'Acqua, P., and Lanzarone, G.A., SLD-Resolution with reflection. *Proc. International Logic Programming Symposium*, Ithaca, New York, November 14–17, 1994.

Boyer, R.S. og Boyer, J.S., The Sharing of Structure in Theorem-Proving Programs. *Machine Intelligence*, vol. 7, s. 101–116, 1972.

Brady, J.M., *The Theory of Computer Science, A Programming Approach*, Chapman and Hall, 1977.

Bratko, I., *Prolog — Programming for Artificial Intelligence.* Second Edition. Addison-Wesley, 1990.

Chomsky, N., On certain properties of formal languages, *Information and Control*, **Vol.** 2, pp. 137–167, 1959.

Christiansen, H., Syntax, semantics, and implementation strategies for programming languages with powerful abstraction mechanisms, *Proc. of 18th Hawaii International Conference on System Sciences*, vol. 2, side 57–66, 1985. (Også *Datalogiske skrifter 1*, Roskilde Universitetscenter, Datalogiafd., 1985).

—, A survey of adaptable grammars, *SIGPLAN Notices*, 25/11, side 35–44, 1990.

Christiansen, H., Deriving declarations from programs, Extended abstract, CPP'97, Workshop on Constraint Programming for Reasoning about Programming, Leeds, April 9-10th, 1997. Elektronisk tilgængelige proceedings `http://www.scs.leeds.ac.uk/hill/cpp`.

Christiansen, H., Automated reasoning with a constraint-based metainterpreter. *Journal of Logic Programming*, 1998a.

Christiansen, H., Implicit program synthesis by a reversible metainterpreter. Fuchs, N.E. (ed.), *Proc. of LOPSTR'97, Lecture Notes in Computer Science*, 1998b.

Chanaud, J.-P., Natural language processing and digital libraries. *Lecture Notes in Artificial Intelligence* 1714, s. 15–31, 1999.

Church, A., The calculi of lambda-conversions. *Annals of Mathematical Studies*, **vol.** 6, Princeton University Press, 1941.

Clocksin, W.F. og Mellish, C.S., *Programming in Prolog*, Third edition. Springer-Verlag, 1987.

Cohen, J., A View of the Origins and Development of Prolog. *Communications of the ACM*, vol. 31, s. 26–36, 1988.

Davis, R., Executable Specifications as Basis of Program Life Cycle. *Proc. of 18th Annual Hawaii International Conference on System Science*, s. 722-733, 1985.

Dewdney, A.K., *The Turing Omnibus*, *61 Excursions in Computer Science*, Computer Science Press, 1989.

Dijkstra, E.W., Notes on Structured Programming. *Structured Programming, A.P.I.C. Studies in Data Processing*, No. 8. Academic Press, 1972.

Earley, J. og Sturgis, H., A formalism for translator interaction. *Communications of ACM*, **vol.** 13, no. 10, side 607–616, 1970.

di Forini, A.C., Some remarks on the syntax of symbolic programming languages. *Communications of the ACM* 6, side 456–460, 1963.

Gazdar, G. og Mellish, C., *Natural Language Processing in PROLOG, An Introduction to Computational Linguistics.* Addison-Wesley, 1989.

Gruska, J., *Foundations of computing*, Thompson Computer Press, 1997,

Gödel, K., Uber formal unentscheidbare Sätze der Principia Matematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38, s. 173–

198, 1931.

Hartmanis, J., Stearns R.E., *Algebraic structure of sequential machines.* Prentice-Hall 1966.

Hill, P.M. og Gallagher, J.P Meta-programming in Logic Programming. Publiceres i bind V af *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press. Pt. tilgængelig som *Research Report Series* 94.22, University of Leeds, School of Computer Studies, 1994.

Hill, P.M. og Lloyd, J.W., *The Gödel programming language*, MIT press, 1994.

Hjelmslev, L., *Sproget, en introduktion.* Berlingske Forlag, 1963.

Kahn, K.M., Pure Prolog in Pure Lisp. *Logic Programming Newsletter*, Winter 83/84, s. 3–4., 1984.

Kamin, S.N., *Programming Languages, An Interpreter-Based Approach.* Addison-Wesley, 1990.

Kfoury, A.J., Tiuryn, J., og Urcyczyn, P., ML typability is Dexptime-complete. *Lecture Notes in Computer Science* 431, pp. 206–220, 1990.

Knuth, D.E., Semantics of context-free languages. *Mathematical Systems Theory* 2, pp. 127-145, 1968.

Kowalski, R.A., The Early Years of Logic Programming, *Communications of the ACM*, vol. 31, s. 38–43, 1988.

Lewis, H.R. & Papadimitriou, C.H., *Elements of the Theory of Computation*, Prentice-Hall, 1981.

Lloyd, J.W., *Foundations of Logic Programming*, Second, extended edition. Springer-Verlag, 1987.

Maier, D. og Warren, D.S., *Computing with Logic, Logic Programming with Prolog.* Benjamin Cummings Publishing Company, Inc., 1988.

McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Communications of the ACM*, **vol.** 3, no. 4, side 184–195,1960.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. og Levin, M.I., *Lisp 1.5 Programmer's Manual*, MIT Press, 1962.

Naur, P. (red.). Revised report on the algorithmic language Algol 60. Communications of the ACM, vol. 10, no. 8., pp. 1–17, 1963.

Penrose, P., *The emperor's new mind: concerning computers, minds, and the laws of physics*, Oxford University Press, 1989.

Pereira, F.C.N. og Sheiber, S.M., *Prolog and Natural-Language Analysis.* Center for The Study of Language and Information, 1987.

Pratt, T.W., *Programming Languages: Design and Implementation.* Second edition. Prentice-Hall, 1984.

Robinson, J.A., A Machine-oriented Logic Based on the Resolution Principle, *Journal of the ACM*, vol. 12, s. 23–41, 1965.

Rosen, S. (red.), *Programming Systems and Languages.* McGraw-Hill, 1976.

Sammet, J.E., *Programming Languages: History and Fundamentals.* Prentice-Hall, 1969.

Sedgewick, R., *Algorithms.* Second Edition. Addison-Wesley, 1988.

Sethi, R., *Programming Languages, Concepts and Constructs.* Addison-Wesley, 1989.

(Sicstus, 1998) *SICStus Prolog user's manual*. Version 3.7, SICS, Swedish Institute of Computer Science, 1998.

Standish, T.A., Extensibility in programming language design. *AFIPS Conference Proceedings*, **vol.** 44, side 287–290, 1975.

Sterling, L. og Shapiro, E., *The Art of Prolog.* MIT Press, 1986.

Sudkamp, T.A., *Languages and machines, an introduction to the theory of computer science*. Addison-Wesley, 1988.

Tanenbaum, A.S., *Structured Computer Organization*, 4th edition, Prentice-Hall, 1999.

Turing, A.M., On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society 2*, no. 42, side 230–265, og no. 43, side 544–546, 1936.

Warren, D.H.D., Implementing Prolog — Compiling Predicate Logic Programs. *D.A.I. Research Report*, nr. 39, 40. Edinbourgh University, 1977.

Wexelblatt, A. (red.), *History of Programming Languages.* Academic Press, 1981.

Wilson, C.B. og Clark, R.G., *Comparative Programming Languages.* Addison-Wesley, 1987.

Winskel, G., *The Formal Semantics of Programming Languages: An introduction* MIT Press, 1993.

Winston, P.H. og Horn, B.P.H., *LISP*, 3rd Edition, Addison-Wesley,1989.

Wirth, N., Program development by stepwise refinement. *Communications of the ACM*, **vol.** 14, side 221–227, 1971.