Programmering som problemløsning

Af

Morten Krog **Morten Jøhnk** **Tore Green**

22. maj 1996

Semesteropgave i Kognitionspsykologi II, foråret 1996

Bifag i Informationspsykologi, 2. semester

Psykologisk Laboratorium, Københavns Universitet Amager

Den til rapporten knyttede vejleder:
Svend Erik Olsen

Resume

Denne opgave handler om de kognitive processer, der finder sted, når en programmør skriver et program. Vi har gennemført fire tænke-højt-forsøg, hvor forsøgspersoner fik en lille programmeringsopgave, som de skulle gennemføre mens de tænkte højt.

Vi gjorde en række interessante observationer under disse forsøg: Forsøgspersonerne var tilbøjelige til at starte på opgaven uden at have analyseret problemet til bunds, hvilket gav dem problemer. De fokuserede meget på det udleverede eksempel, og havde derfor svært ved at se en god måde at overskue problemet på. Det var tydeligt, at programmørerne opbyggede en mental model af deres program, som de brugte som reference gennem hele opgaveløsningen. Endelig omtaler vi begrebet "pæn programmering", som tilsyneladende var væsentligt for forsøgspersonerne – dog ikke på en sådan måde, at de nødvendigvis skrev pæne programmer.

Indholdsfortegnelse

1. Forord

Denne rapport har to målgrupper – programudviklere og (kognitions)psykologer. Programudviklere vil primært være interesserede i de konklusioner og anbefalinger, vi kommer frem til. Disse vil også være interessante for kognitionspsykologer, men for den gruppe er det nok hele rapporten som sådan, der er interessant. I rapporten vil vi anvende en del fagtermer hentet fra udviklernes verden (primært forskellig notation for matematiske udtryk), hvis betydninger måske vil være ukendte for de fleste andre læsere af rapporten. For at løse dette problem har vi skrevet et appendiks (se side 17) med en indføring i de begreber, som læsere, der ikke har en baggrund som programudviklere el. lign, kunne have brug for at få forklaret.

2. Indledning

Udvikling af programmer til datamater er en krævende proces, der stiller store krav til udviklernes evner og viden. Det er en opgave, der bygger på udviklerens kognitive evner, og er derfor interessant at studere ud fra en kognitionspsykologisk indgangsvinkel. For os er der også andre grunde til, at vi vælger at beskæftige os med dette emne, da vi er datalogistuderende og formentlig vil komme til at have programudvikling som en væsentlig del af vores arbejde.

I denne opgave vil vi studere, hvilke kognitive processer der aktiveres som en del af udviklingsarbejdet. Vores fremgangsmåde vil primært være empirisk, da vi vil stille en mindre udviklingsopgave til en række datalogistuderende og observere, hvorledes de griber den an.

Hvad er det så for krav til udviklerens kognitive apparat, en udviklingsopgave stiller? For at besvare dette må vi først skitsere, hvorledes arbejdet med en udviklingsopgave normalt forløber. Udgangspunktet er et eller andet problem, hvor nogen har den ide, at det ville være smart med et program, der kunne tage sig af det. Problemdomænet vil i større eller mindre grad være ukendt for udvikleren[1], som modtager en kravspecifikation, der er en beskrivelse af problemet og den ønskede løsning. Ud fra kravspecifikationen laver udvikleren et design, der er en ny beskrivelse af det ønskede program – denne gang i form af en mere konkret beskrivelse af, hvorledes programmet skal laves. Den næste fase i projektet er implementationsfasen, hvor selve programmet skrives. Til sidst skal det testes at programmet lever op til kravene, og fejlene[2] skal rettes.

Ovenstående er den struktur af udviklingsprojekter, som datalogistuderende undervises i. Det lyder jo meget let, men der er alvorlige problemer: Kravspecifikationen er normalt udformet i domænespecifikke termer, og er sjældent særlig præcis: Der er tit en mængde underforstået viden, og der er ofte ønsker og krav, som først opdages senere i projektet. Opdelingen i faser kan ikke gennemføres i praksis, da fejl, når de opdages, ofte vil medføre ændringer til designet, og programmer implementeres og afprøves i små bidder for at øge overskueligheden af opgaven. En

[1] Som regel vil der være tale om en gruppe af udviklere med hvert sit speciale, men vi vil for nemheds skyld omtale det, som om der er netop én udvikler.

[2] Ethvert program har fejl.

udviklingsopgave stiller således krav til udvikleren, om at han hurtigt skal kunne tilegne sig den nødvendige domænespecifikke viden, at han skal kunne gennemskue uklarheder, mangler og selvmodsigelser i kravspecifikationen, at han skal kunne overskue alle konsekvenser af de beslutninger han træffer i designfasen, at han skal kunne omsætte sit abstrakte design til et konkret program uden fejl, at han skal kunne lave en afprøvning uden at blive påvirket af, at han selv har skrevet programmet, og måske slet ikke ønsker at finde fejl, og at han skal kunne overskue sit program i så høj grad, at han er i stand til at lokalisere og rette fejl – uden at introducere nye fejl.

Vi har ikke mulighed for at undersøge rigtige udviklingsprojekter, da de som nævnt er meget langstrakte. Derfor vælger vi selv at foretage en række små forsøg, hvor vi kan observere udviklere i arbejde i et kortere tidsrum. Dette afgrænser os fra at studere effekter, der bunder i udstrækningen og kompleksiteten af 'rigtige' udviklingsprojekter, men vi mener, at vi vil se mange af de samme kognitive processer i spil. Konkret vil vi stille en mindre programmeringsopgave til nogle forsøgspersoner og bede dem tænke højt mens de løser den.

3. Den stillede opgave

I dette afsnit vil vi redegøre for den stillede opgave, måden opgaven blev stillet på og de omstændigheder, hvorunder forsøgene foregik.

Selve opgaven er en lille programmeringsopgave, hvis løsning gennemgås i afsnit X. Inden forsøgspersonerne fik stillet selve opgaven, læste de en kort introduktion, hvoraf det fremgik, at opgaven var at udvikle et lille program, hvor de skulle 'tænke højt' mens de løste opgaven. Så blev selve opgaven præsenteret og løst, hvorefter vi interviewede forsøgspersonerne. Opgaveteksten var:

Opgaven:

Matematiske udtryk kan opstilles på mange forskellige måder. Den mest almindelige til daglig brug er infix notation – f.eks.:

```
2+2
```

For at få en bestemt, ønsket struktur i et udtryk kan der indsættes parenteser. Faktisk kan der sættes vilkårligt mange parenteser i et udtryk:

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

Ovenstående udtryk kan også skrives som:

```
2*3+1
```

Din opgave er at udvikle et program, der kan udskrive et udtryk med kun de parenteser, der er nødvendige under de gængse regler. Dit program skal ikke være i stand til at indlæse udtryk – de udtryk, du ønsker at køre programmet på, må gerne være en fast del af selve programmet.

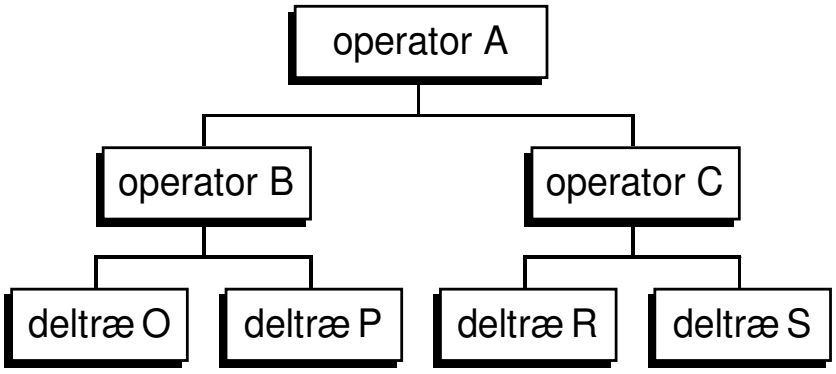
Dit program skal kunne håndtere de fire almindelige regningsarter +, -, * og /. Det er ikke meningen, at dit program skal kunne håndtere negative tal eller monært -, og du kan bruge heltal eller flydende tal efter behag. Dit program må ikke ændre udtrykkenes struktur.

Det centrale i opgaven er at udskrive et matematisk udtryk med et minimum af parenteser. Opgaveteksten giver et eksempel på et udtryk med mange parenteser, hvor alle parenteserne kan fjernes, uden at dette ændrer udtrykket. For at løse problemet, har forsøgspersonerne brug for at formulere et sæt regler om, hvornår parenteser er nødvendige. Ved at give et eksempel, hvor det slet ikke er nødvendigt med parenteser, ungår vi at eksemplet leder forsøgspersonerne på vej. Da vi ønsker at studere forsøgspersonernes problemløsning og ikke deres programmeringsfærdigheder, er opgaven formuleret, så forsøgspersonerne kan gå direkte til selve opgaven, uden at skulle programmere indlæsning af vilkårlige matematiske udtryk, men i stedet kan lade de udtryk, de ønsker at prøve deres program på, være en del af programmet.

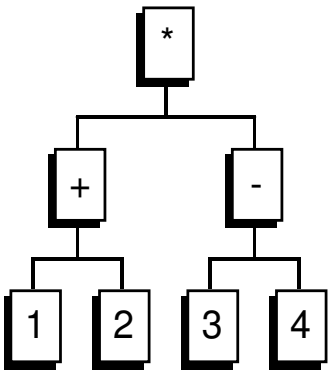
3.1 Løsning af opgaven

Dette afsnit beskriver det, vi betragter som en god løsning af opgaven. Det er langt fra den eneste måde at løse den på, men det er den løsning, vi havde forventet at se fra forsøgspersonerne. Forklaringen af løsningen bygger på begreberne præsenteret i appendikset (fra side 17).

Opgaven lød, at man skulle tage et udtryk og lave et program — eller stille de regler op udfra hvilke man kan skrive et program — således at man kunne tage et matematisk udtryk og skrive det om til et infixudtryk med netop det antal parenteser, der er nødvendig for at start- og resultatudtrykket er ens (men ikke nødvendigvis identiske). Der er ikke nødvendigvis noget krav om, at man skal starte med et infixudtryk, så det første man bør gøre sig klart er, hvorledes man vil tænke på det udtryk man starter med.



Figur Fejl! Ukendt argument for parameter.**: toppen af et træ**



Figur Fejl! Ukendt argument for parameter.**: Eksempel på et udtryk af formen fra figur 6**

Når der skal afgøres om der skal parenteser omkring et deludtryk som R C S, skal man kende Cs præcedens, As præcedens og Cs associativitet. For nemt at kunne referere dem, er det praktisk at tænke på dem som et træ.

Står man med tilfældet i **Fejl! Ukendt argument for parameter.** og vil afgøre om der skal parenteser omkring B og Bs operander og C og Cs operander, da skal man kende A, B og Cs præcedens. Ser man et eksempel på et infixudtryk på formen fra **Fejl! Ukendt argument for parameter.** (**Fejl! Ukendt argument for parameter.**) som 1+2*(3-4): Det er ret oplagt at parentesen i udtrykket er nødvendig, men hvilken regel kan man opstille som det kan udledes af? Hvis *s præcedens er højere end +s, da vil

operatoren binde 2 stærkere end + vil, så derfor skal der parenteser rundt om operatoren + og dens operander. Det samme gør sig gældende for *s forhold til -. Hvis operatoren A havde samme præcedens som B (f.eks. hvis udtrykket var (1+2)+3), da er parenteserne ikke nødvendige.

Da man læser og beregner infixudtrykket fra venstre mod højre, kan problemet omkring operatorernes associativitet kun opstå i det højre deltræ af **Fejl! Ukendt argument for parameter.**. Hvis operatorerne A og C således har samme præcedens, skal der stadigvæk parenteser omkring det højre deltræ hvis operatoren A ikke er associativ (dvs. hvis det er - eller /).

Kernen i denne løsning er, at man ikke alene skal tænke på en operator og dens operander. Det er også nødvendigt at tænke på denne del som operand til en anden operator. Man skal således kende præcedens i to niveauer i udtrykket og desuden tænke på, at - og / ikke opfører sig som + og *.

Løsningen her kan nemmest findes hvis man hele tiden tænker på udtryk som træer. Gør man ikke det, skal man alligevel lave det arbejde, der giver dig mulighed for at referere til deludtryk på samme måde, som vi her har manipuleret deltræer.

4. Forsøgenes forløb

I det kommende afsnit vil vi kort gennemgå de fire forsøg vi har foretaget. Hver forsøgsperson er en rimelig rutineret programmør med mindst tre års uddannelse i datalogi bag sig. Med udgangspunkt i deres uddannelse og deres erfaring burde det være nemt for dem at løse opgaven. De 2 første forsøg blev afviklet på cirka en time. Det 3. tog cirka 1½ time og det 4. cirka 35 minutter.

Ved starten af forsøget blev forsøgspersonen sat ned foran en terminal med en stak bøger forsøgspersonen kunne bruge efter behag. Den 2. forsøgsperson brugte dog aldrig terminalen, men udførte hele arbejdet på et stykke papir.

Til de to sidste forsøg havde vi besluttet at hjælpe dem i gang med at tænke på problemerne i opgaven, hvis de ikke selv umiddelbart så dem. Denne beslutning blev taget på baggrund af, at de to første forsøgspersoner havde haft meget svært ved at spore sig ind på den løsning, vi havde forestillet os, og vi havde opgaveformuleringen mistænkt for at være en del af årsagen. Det bedste i denne situation vil nok have været at gennemføre forsøgene, og derefter foretage en ny forsøgsrække med en omformuleret opgave. På grund af det fremskredne tidspunkt i vores opgaveperiode valgte vi dog i stedet at bibeholde opgaven og være lidt flinkere til at hjælpe med præciseringer og vink undervejs. Det blev kun nødvendigt i det ene af de to tilbageværende forsøg.

Beskrivelsen herunder er et kommenteret referat af hvorledes de 4 forsøg gik.

4.1 Klaus

Klaus starter med at se på eksemplet i den udleverede opgave. Han fokuserer på det *at fjerne* parenteser. Da det umiddelbart ser ud til at være nemt at fjerne parenteser omkring et enkelt tal, starter han med at overveje hvorledes det kan gøres. For at gøre det nemt for sig selv beslutter han, at der ikke er nogen grund til at arbejde med tal på mere end et ciffer.

Klaus er allerede i gang med programmeringen her og bliver hurtigt i tvivl om en teknisk detalje, som han forsøger finde svaret på i en bog.

Det går op for Klaus, at hele opgaven er større end han først havde antaget, og han beslutter derfor at, det at fjerne parenteser omkring et enkelt tal, skal uddelegeres i en opgave for sig selv og ikke er en central del af opgaven.

Hans arbejdsproces indtil nu (ca. 20 min.) virker som om, han arbejder med denne række af tegn og har tænkt sig at stille nogle regler op, som han vil bruge til at filtrere rækken med. Hver regel skulle så fjerne nogle parenteser.

Når der stødes på tekniske problemer med at programmere et eller andet, så bruges der ikke tid på at finde ud, af hvad der er galt med det, han har gjort. I stedet laves det på en radikal anderledes måde.

Efter cirka en ½ time er det lykkedes Klaus at skrive et program der fjerner vilkårligt mange parenteser omkring et ciffer. Så går det op for ham, at det næste problem nok kræver lidt mere tankearbejde og går over til at gøre noter på et stykke papir.

Han gør sig det klart, at nogle operatorer kræver parenteser og andre gør det ikke. Hans idé er at kontrollere, om der inden i en parentes står en operator, der kræver parenteser. Hvis der ikke gør det, så vil han fjerne parenteserne.

Efter i alt 35 minutter genkender han strukturen i matematiske udtryk: Et udtryk er enten et tal, eller to udtryk med en operator imellem. Han opgiver næsten alt det han har lavet indtil nu. Det viser sig, at han har genkendt den del af udtrykkes struktur, der stammer fra parenteserne, men endnu ikke rigtig den del, der stammer fra operatorernes præcedens.

Der bliver brugt meget tid på at skrive ting ind, og Klaus er tilsyneladende meget gladere for sin nye ide: "Det er meget pænere end det var før!", siger han.

Efter 50 min. synes han, at han har lavet en del og prøver, om det virker med eksemplet fra opgaveteksten. Det gør det, ikke og han bliver i tvivl om, hvorvidt han har haft fat i de rigtige kriterier for at fjerne parenteser.

Til sidst får han det eksempel til at virke, men nu er der en parentes tilbage om hele udtrykket [f.eks. (1+2+3)]. Han synes det er et specialtilfælde og finder på en ting der kan opdage netop denne type parenteser og fjerne dem.

Efter forsøget mener han, at han ikke brugte nok tid på at tænke sig om, inden han gik i gang med at lave noget. Det kommer lidt bag på ham, at han også skulle tage højde for operatorernes associativitet, men han kan godt se det, da vi giver ham et par eksempler. Klaus mener det er meget typisk for hans arbejdsmetode, at han bare kaster sig ud i det.

4.2 Frederik

Frederiks første tanke er, at lave udtryk til postfix notation og så derefter lave dem tilbage. Det opgives dog hurtigt igen. Dernæst er tanken at lave et Prolog-program, der laver omskrivningen. Programmeringssproget Prolog blev overvejet fordi, det her er meget nemt at lave programmer udfra logiske regler. Hans idé var at lave noget mønstergenkendelse. Efter ca. 10 minutter har Frederik fat i, at disse mønstre skulle have noget med præcedensregler for operatorer at gøre.

Efter ca. et kvarter tager han fat i en bog, for at få afklaret nogle tekniske detaljer og cirka 5 minutter senere udbryder han: "Jeg mangler at have gjort mig klart, hvor der egentlig skal være parenteser. Der skal omkring udtryk med plus, men ikke omkring udtryk med gange.".

Der går ca. 5 minutter med ingen ting. Frederik synes ikke, det ser ud til at være godt. "Det ligner noget, der ikke er godt. Hvordan kan man huske, hvor der var en parentes?".

Der overvejes om parenteser kan opfattes som en slags operatorer, men ideen opgives igen. Der er stillet nogle få regler op på dette tidspunkt (der er gået cirka. 40 minutter) og nu vil han se om de regler virker. Der tænkes nu på matematiske udtryk som træer. Det er gået op for Frederik, at når man har en * operator i roden, og en + operator i venstre deltræ, da skal man have parenteser omkring venstre deltræ [eks. (3+2)*4 er ikke det samme som 3+2*4].

Det programmeringssprog han nu anvender synes han er pænt, i modsætning til andre programmeringssprog.

Frederik har tilsyneladende svært ved at opgive de parenteser, der er i vores eksempler, på trods af, at der tydeligt står i opgaveformuleringen, at man ikke nødvendigvis skal skrive udtryk op på netop den måde de er i opgaveteksten. Forsøgslederen hjælper et par gange for at overvinde denne modstand, da opgaven ellers bliver noget større end vi havde tænkt os, og forsøget vil tage for lang tid.

Efter cirka en time er forsøget færdigt. Frederik sagde bagefter, at han troede man ofte støttede sig meget til det kursus man fulgte, når man skulle løse en opgave. Således vidste man jo nogenlunde hvad kurset forventede man gjorde.

4.3 Rasmus

Det første Rasmus gjorde var, at spørge om det var meningen at man skulle fjerne overflødige parenteser. Da vi havde besluttet at hjælp de to sidste forsøgspersoner til at finde den rigtige struktur at arbejde med, svarede vi, at det var meningen, at man skulle sætte de nødvendige parenteser. Rasmus overvejede hvilke parenteser, der var nødvendige og blev enig med sig selv om, at det var operatorpræcedens, der afgjorde om en given parentes var nødvendig.

Rasmus besluttede sig for at betragte alle udtryk som postfix udtryk [dvs. 3+4 bliver skrevet som 3 4 + og (3+4)*2 som 3 4 + 2 *]. Han havde overvejet at betragte udtryk som træer, men havde forkastet det [efter forsøget fremgik det, at han forkastede den løsning ud fra en vurdering af hvad der var nemmest at lave].

Efter at have valgt anskuelsesvinkel skrev han et par eksempler ned. Derefter blev han i tvivl om hvad opgaven egentlig gik ud på og så i opgaveteksten.

Nogle løse tanker blev til en generel metode til at manipulere udtryk med, som byggede på en metode, Rasmus havde set i undervisningen på datalogi. Han var efterfølgende overrasket over, at det havde taget ham så lang tis at komme i tanke om metoden. For at gøre programmet simplere, antog han, at tal kun havde ét ciffer. Der var på nuværende tidspunkt gået cirka 20 minutter og han var begyndt at skrive på et program. Der var tekniske vanskeligheder og der blev tit forsøgt at køre programmet, selv om det var hverken helt eller halvt færdigt. Han brugte de fejlmeddelelser han fik ud af sine forsøg, til at lokalisere problemerne i sit program. Det var dog ikke særligt effektivt for ham.

Efter 25 minutter begyndte han at konkretisere sin løsningsskitse i forskellige delfunktioner. Det var egentlig før han havde fundet ud af hvad han ville med delfunktionerne.

"Nu står vi med en gange- eller divisionsoperator og så skal vi undersøge om det forrige er + eller -…", sagde han og sprang frem og tilbage i det program han var ved at skrive.

Da der var gået 35 minutter udbrød han "er det ikke noget slamkode, det her". Han var tydeligvis utilfreds med det han havde lavet og han havde desuden lige opdaget endnu et tilfælde, der krævede parenteser. Han overvejede at opgive det han havde lavet ind til nu.

Selvom han ikke er færdig med at overveje hvornår der skal sættes parenteser, går han i gang med at lave en funktion, der sætter parenteser de steder hvor han indtil nu har opdaget der skal være nogen.

Efter 45 minutter prøver han igen at køre sit program, der ikke *vil* køre. Der ledes efter fejl i lang tid. I udskriften fra programkørslen dukker der et 0 op, som han bruger meget tid på at undre sig over. Han har helt glemt, at han for noget tid tilbage selv bad programmet om at skrive det 0 ud. Efter lang tid siger han "jeg prøver at finde en strategi til at finde fejlen".

Så prøver han med et eksempel på papir og skriver blandt andet det 0 op, uden at tænke nærmere over det. Derefter konkluderer han, at det ikke er muligt for 0'et at dukke op. Først da han igen kigger på programmet opdager han hvorfor det 0 kom ud. "Det er lige før man tror det virker", siger han og anvender programmet på et problemløst eksempel.

Det lykkedes ikke Rasmus at opdage problemet omkring operatorassociativitet under forsøget, selv om han sagtens kunne se, at det udgjorde et problem, da vi efter forsøget gjorde ham opmærksom på det.

4.4 Jesper

Jesper læser opgaveteksten og siger umiddelbart: "Det er jo et simpelt træ. Der er ingen grund til at have parenteserne med, da man direkte ud fra træet kan rekonstruere de nødvendige parenteser". Derefter checker han opgaveteksten for at se om han virkelig må det.

Derefter konstaterer han, at man for en given operator, skal man undersøge, om der skal parenteser omkring udtrykkene på hver sin side af operatoren [herved adskiller han sig fra de 3 andre, der *næsten* udelukkende overvejede om der skulle parenteser omkring operatoren og dens operander].

Der kan kun være to ting i et udtrykstræ: tal og operatorer. "Lad os lave det generelt. Der er ingen grund til ikke at lave det generelt: Det ville være dårlig stil.".

Jesper beslutter sig til at * og / skal have præcedens 1 og + og - præcedens 0. Højeste præcedens medfører parenteser omkring operander med lavere præcedens (der er nu gået 10 minutter). Derefter overvejer han om også tal skal have en præcedens. Man kan jo give tal den højeste præcedens, så får de aldrig parenteser om sig. Jesper synes det er en meget pænere løsning.

Efter 15 minutter har han allerede skrevet en stor del af programmet og giver sig tid til at ændre i det, "for at gøre det pænere". Jesper ser på sit program og siger: "Vi tager overhovedet ikke højde for associativitet… Det gør heller ikke nogen forskel".

Han prøver med et simpelt eksempel: 2+2, men ændrer det til 2+3, for ellers kan man ikke se forskel på de to 2-taller. Derefter vil han afprøve om der sættes parenteser korrekt og prøver med 3*(2+2) og det virker tilsyneladende. Han siger lidt for sjov: "Det er måske ikke en komplet afprøvning i DIKU-stil".

Efter 25 minutter prøver han med et eksempel, og inden har lader programmet prøve, siger han højt hvad han forventer som resultat. Hurtigt derefter prøver han med endnu et eksempel 30-(2+3). Det bliver til 30-2+3 og afslører dermed, at der skulle tages højde for operatorassociativitet. Der programmeres lidt og derefter prøves eksemplet igen og nu virker det. Derefter checker han det ark papir, opgaven stod på, og konstaterer at han er færdig.

Efter forsøget siger Jesper, at han hurtigt var klar over, at der var to niveauer af operatorer. Desuden syntes han meget af opgavens løsning var oplagt ud fra de kurser man tog på DIKU.

4.5 Opsamling på forsøgene

Som nævnt indledningsvis var vi overrasket over hvor svært, forsøgspersonerne havde ved at løse opgaven. Der var forskel på deres angrebsvinkel, men også mange fælles træk – især var det de samme problemer, der gik igen for de første tre forsøgspersoner. Jesper skilte sig ud fra de øvrige ved at bruge mindre tid på opgaven og faktisk få et korrekt program ud af det. Han så fra starten den løsning, som vi anser for den pæneste, og det var uden tvivl dette, der hjalp ham igennem. Retfærdigvis skal det siges, at Jesper var den mest erfarne programmør blandt forsøgspersonerne, da han i nogle år har haft erhvervsarbejde som programmør.

5. Resultater

Vi har valgt at beskæftige os med fire fænomener, som vi observerede ret tydeligt i vores forsøg. Det første fænomen er valg af løsningsstrategi – eller rettere manglen på bevidst valg af løsningsstrategi. Det andet fænomen, vi vil beskæftige os med er fiksering. Vi vil se på forsøgspersonernes tendens til at have et indre billede (eller en mental model, om man vil) af programmet, og endelig vil vi omtale begrebet "pæn programmering", der tilsyneladende spiller en vis rolle for forsøgspersonerne.

5.1 Løsningsstrategi

Der findes et antal forskellige modeller for, hvordan menneskelig problemløsning foregår, og hvordan den bør foregå. Fælles for en række af disse er, at de består af en række faser: F.eks. "Problemforståelse – planlægning – udførelse – tilbageblik" (Polya) eller "Identificer – definer – eksplorer – ager – led/lær" (Bransfords IDEAL-model). På baggrund af disse modeller finder vi opførslen hos vores forsøgspersoner interessant.

Alle fire forsøgspersoner er trænet gennem tre eller flere år på datalogistudiet, og en af de ting, der lægges vægt på i undervisningen er problemanalyse: Dét, at analysere sig frem til løsningen inden man giver sig til at implementere den. Forsøgspersonerne forbryder sig imod dette i forskellig grad, men det er bemærkelsesværdigt, at ingen af de fire formulerer korrekte og dækkende regler for parentes-sætning, inden de giver sig i kast med selve programmeringen. Som vi ser det, er der især to årsager til dette:

- Der er tendens til at angribe det første delproblem, der dukker op, frem for at analysere til bunds.
- Når én regel er formuleret (om operatorernes præcedens), gøres der ikke en indsats for at afkræfte, at denne regel er tilstrækkelig.

Den anden årsag har bl.a. med fiksering at gøre, og vi udsætter således diskussionen af den til afsnit 5.2.

En meget anvendt strategi ved løsning af større problemer er "del og hersk" (eng. *divide and conquer*) (Sacerdoti, 1977, refereret i Pennington og Grabowski, 1990). Denne strategi er især fremtrædende indenfor datalogien, da opdeling i mindre delproblemer er den eneste måde at opnå et program, der er til at overskue. Pennington og Grabowski (1990) mener, at en variant af strategien er nødvendig, når der er tale om programmering: De delproblemer, problemet opdeles i, afhænger på nogle punkter af hinanden. Der er derfor ikke tale om en fuldstændig opdeling, men om det, Hayes-Roth kalder *opportunistic planning* (Pennington og Grabowski, 1990). Grundprincippet med opdeling i delproblemer er imidlertid det samme.

Programmeringssprog understøtter denne opdeling ved at give mulighed for såkaldte underprogrammer[3]; Et underprogram kan løse et delproblem, og kan kaldes af det egentlige program på passende steder.

Trænede programmører leder efter ting, der kan generaliseres. En generalisering, som alle fire forsøgspersoner hurtigt foretog var at tale om + og *, men ikke om - og /. I dette tilfælde ikke nogen heldig generalisering, fordi + og - hhv. * og / godt nok har samme præcedens, men ikke samme associativitet. Normalt er generaliseringer imidlertid gode, fordi man ofte kan skrive ét underprogram, der løser flere delproblemer, hvis disse delproblemer har fælles træk, og derved kan man spare noget arbejde. Et fælles træk ved vores forsøgspersoners brug af underprogrammer var, at de giver mulighed for effektivt at dele op i delproblemer: Når man møder et rimeligt afgrænset delproblem, som kan løses uafhængigt at resten af problemet (f.eks. "Det er nu sikkert, at der skal sættes parenteser om dette udtryk – sæt dem!"), løses dette delproblem af et underprogram. Enten forlader forsøgspersonen det sted i programmet, han arbejder på lige nu, for at vende tilbage når delproblemet er løst, eller også antager han blot, at dette delproblem kan løses af et underprogram, og skriver så underprogrammet, når han er færdig med den aktuelle opgave.

Den rette måde at anvende "del og hersk"-strategien lægger sig op ad IDEAL-modellen: Oversku problemet, opdel i passende delproblemer, og løs dem et ad gangen. Dette øger også muligheden for at opdage den generalitet, der evt. måtte være, sådan at man slipper for at løse beslægtede opgaver flere gange. Vi ser hos forsøgspersonerne en tendens til i stedet at løse det første overskuelige delproblem, de får øje på, i stedet for først at foretage hele opdelingen, og derefter give sig til at løse delproblemerne efter tur.

Det tydeligste eksempel på dette fænomen ser vi hos Klaus. Han tager fat på det simpleste delproblem, der falder ham ind efter at have set på opgaven: Parenteser om tal (som i "(2)+(2)") er overflødige og kan fjernes. Da han har løst dette delproblem, går han videre til at forsøge at finde flere regler for, hvornår parenteser kan fjernes – og det viser sig, at en mere generel regel dukker op, der indholder parenteser om tal som specialtilfælde. Dette får den konsekvens, at han efter ca. 30 minutter ikke længere har brug for det første arbejde, hvorfor det smides væk.

I en vis grad bliver forsøgspersonerne selv opmærksomme på, at deres analyse ikke er komplet – både Frederik og Klaus konstaterer efter ca. 30 minutters forløb "Jeg mangler stadig at gøre mig helt klart, hvornår der skal være parenteser". Man kan dog sætte spørgsmålstegn ved, hvor dybt denne konstatering er forarbejdet, idet ingen af dem efter dette udsagn faktisk formulerer egentlige regler – heller ikke en præcis formulering af den regel om præcedens, som de tilsyneladende begge arbejder ud fra. Hvis de havde forstået, at det var en klar mangel ved deres opgaveløsning, at de endnu ikke havde præciseret den opgave, de skulle løse, ville de nok have stoppet op og foretaget denne præcisering.

Det er sandsynligt, at tendensen til ikke at tænke hele problemet igennem blandt andet skyldes opgavens størrelse. Hvis forsøgspersonerne fik en større opgave, som forventedes at tage lang tid at løse, ville de formentlig foretage en grundigere analyse end i denne ret lille opgave. Ud fra opgavens størrelse kunne man forestille sig, at de følte, at de på forhånd burde kunne overskue opgaven, og derfor ikke ønskede at bruge tid på at definere den nærmere – tre af forsøgspersonerne gav imidlertid udtryk for, at de *ikke* havde det store overblik fra starten. En anden og lignende forklaring på den noget hovedkuls start på opgaveløsningen er, at de (bevidst eller ubevidst) ikke ønskede at være kørt fast i længere tid i et forsøg, hvor der sad tre medstuderende og betragtede dem – det må lægge et vist pres for at komme i gang så hurtigt som muligt.

---

[3] også kaldet procedurer, funktioner eller på engelsk *subroutines*

5.2 Fiksering

De tre første forsøgspersoner brugte meget tid og energi på at overveje hvorledes man *fjerner* parenteser fra udtrykkene. Dette var helt tydeligt centralt for deres fremgangsmåde i opgaven. I det eksempel, der er udleveret med opgaven er der jo netop også "for mange" parenteser med. Dunckers forsøg (Spelling, 1972) viser, at forsøgspersoner ofte støtter sig meget til illustrationer i opgaveformuleringen og det kan vi genkende her. Det vigtige er at genkende, at man kan se på udtryk på andre måder end i infixnotation med parenteser. Den sidste forsøgsperson ser hurtigt denne forskel og kommer dermed helt over problemet med illustrationerne.

En følgeproblematik er, at forsøgspersonerne opfatter udtrykkene som de er skrevet op, i stedet for at se dem som den abstraktion, de repræsenterer. De hænger sig mindre i den detalje end i parenteserne, men dette hænger muligvis sammen med at man under alle omstændigheder er nød til at opbygge en abstrakt repræsentation af matematiske udtryk for overhovedet at kunne arbejde med dem i et program.

De alment kendte regler om hvornår der skal sættes parenteser udmøntes hurtigt i, at operatorpræcedens er den styrende regel. Da forsøgspersonerne hurtigt ser denne regel og da de hurtigt kan konstruere nogle eksempler, hvor operatorpræcedens korrekt afgør om der skal sættes parenteser, forsøger forsøgspersonerne ikke at overveje om der kan være andre regler (f.eks. som operatorassociativitet). De eksempler forsøgspersonerne stiller op tjener heller ikke til at afvise, at operatorpræcedens er den eneste regel – hvilket ikke er overraskende, da mennesker generelt er dårligere til at afvise hypoteser end til at bekræfte dem (jvf. Wasons kortforsøg o.lign, se f.eks. Smyth m.fl. kap. 12).

Jesper er den eneste forsøgsperson, der højlydt tænker over hvorledes man kan forvente det program han havde skrevet ville virke og i nogen grad overvejede at konstruere eksempler, der ville afprøve hans regel, men han gjorde det heller ikke konsekvent.

5.3 Det indre billede

Det er ret tydeligt for os, at forsøgspersonerne udover at skrive tekst på papiret eller på skærmen samtidig udviklede deres egen "interne repræsentation" af programmet. Faktisk virker det rimeligt at påstå, at programmering består i at opbygge en løsning af problemet inde i hovedet på programmøren, og derefter (eller undervejs) formulere denne løsning i det programmeringssprog, man har valgt. Denne opfattelse svarer til Jonhson-Lairds brug af begrebet "mentale modeller" (Smyth m.fl., 1994, p.232 og 347).

Der er flere tegn på, at der opbygges en sådan mental model undervejs. Forsøgspersonerne viser en klar evne til at skifte mellem forskellige abstraktionsniveauer eller detaljeringsniveauer, når de læser deres eget program. Når behovet opstår for at kigge på en del af programmet igen (f.eks. for at genopfriske hukommelsen eller for at lede efter fejl), ser man på forskellige dele af programmet på forskellig måde, alt efter relevansen for det, man leder efter. For at give et simpelt eksempel[4]: Sætningen

```
for(i=0;i<=laengde;i++) printf("%s\n",tabel[i]);
```

kan efter behag læses som "Tæl med variablen[5] *i* fra 0 op til *laengde*, for hver værdi af *i* udskriv den tilsvarende plads i tabellen *tabel* efterfulgt af et linieskift" eller blot som "udskriv indholdet af *tabel*".

[4] og med fare for at forvirre ikke-programmeringskyndige læsere

[5] En variabel opbevarer en værdi (f.eks. et tal, et stykke tekst el.lign.), der kan bruges eller ændres rundt omkring i programmet. Man kan tænke på den som en kasse med variablens navn stående udenpå og en seddel med værdien

Denne gruppering understreges ofte med indrykning, tomme linier og andre former for visuel gruppering af teksten i programmet, men den må bunde i en forståelse af programmet, repræsenteret i programørens mentale model.

Denne evne til at skifte detaljeringsniveau efter behag gør det muligt at overskue programmet og danne sig overblik, men kan til tider være en forhindring, når man leder efter fejl i et program. Ofte vil programmøren have en ide om, hvor fejlen skal findes, og han vil derfor fokusere på dette sted i programmet, og betragte andre dele af programmet mere overfladisk. Hvis fejlen viser sig at være et andet sted end ventet kan han have set direkte på fejlen flere gange, men uden at forarbejde det sete så dybt, at han opdager fejl. Rasmus brugte lang tid på at finde en fejl af netop denne type: Hans program udskrev "0" i stedet for "(3+1)*2", og han ledte derfor efter en fejl i programmet. Undervejs i denne fejlfinding skrev han flere gange det famøse "0" på sit papir, uden at indse, at der slet ikke burde indgå et "0" i eksemplet, og at fejlen lå her og ikke i selve beregningen.

Et andet tegn på, at programmøren har en mental model af programmet var den måde, rettelser og tilføjelser i programmet foregik på. Meget ofte førte rettelser ét sted i programmet til rettelser et helt andet sted. Det mest typiske eksempel på dette var brugen af variable: Når man et sted i programmet anvender en variabel sker det ud fra en forventning om, at den indeholder en bestemt værdi – f.eks. forudsætter eksemplet ovenfor, at *laengde* indeholder antallet af elementer i tabellen *tabel*. Det var almindeligt blandt vores forsøgspersoner, at når en variabel blev anvendt, opdagede programmøren, at den ikke havde fået den værdi et andet sted i programmet, som han nu forventede. I vores eksempel kunne det tænkes, at der et sted blev ændret på størrelsen af *tabel*, uden at *laengde* blev ændret, og programmøren ville så komme i tanke om dette, når han skulle bruge værdien af *laengde*.

Denne form for opdagelse må komme af, at man associerer variablen med dens værdi, og værdien må ligge repræsenteret i programmørens mentale model af programmet. Klaus sagde direkte i vores interview efter forsøget, at han hele tiden havde et billede af programmets "tilstand" inde i hovedet, men vi ser klare tegn på, at dette også var tilfældet hos de andre forsøgspersoner.

5.4 Æstetik i programmering

Alle fire forsøgspersoner talte undervejs om, at deres løsning var "pæn" eller mindre pæn. Disse udtalelser dækker over, at de har en form for fornemmelse for, hvordan et program bør være. Det er i høj grad denne fornemmelse, der styrer, hvilken retning deres tanker tager, men det betyder ikke, at de nødvendigvis skriver pæne programmer.

Begrebet "pæn programmering" er i høj grad et begreb, man stifter bekendtskab med på datalogistudiet. De fleste datalogiske problemer har mange forskellige mulige løsninger, og en del af studiets indhold er selvfølgelig at lære folk, hvilke løsninger de bør vælge. Frederik kategoriserede desuden forskellige programmeringssprog i pæne og mindre pæne – tilsyneladende efter, hvor let det er at skrive pæne programmer i de forskellige sprog. Når vi bad forsøgspersonerne forklare begrebet "pænt" nærmere, var

---

indeni. Når man får brug for det, kan man så enten finde den rette kasse og læse, hvad der står på sedlen, eller man kan lægge en ny seddel med en ny værdi i kassen.

det ord som "struktureret"[6] og "generelt", de brugte – vores erfaring siger os, at "effektivt" og "læseligt" også hører med.

På grund af den udbredte brug af disse begreber på studiet kunne det have været interessant at undersøge nærmere, hvor ens eller forskellig de fire forsøgspersoners opfattelse af begrebet var, og hvor vidt der var tale om en egentlig grundlæggende tilegnelse af begrebet, eller de blot gentog deres underviseres ord. Dette falder dog lidt ved siden af vores undersøgelse af programmering som problemløsning, så det har vi ikke set nærmere på i denne omgang.

Ved nærmere eftersyn viser pæn programmering sig, at have spillet en anden rolle i forsøgene end man kunne forvente. Tre af forsøgspersonerne (Klaus, Frederik og Rasmus) kom alle under forsøgene med negative udbrud i stil med "Nej, hvor er det noget slamkode det her!". Til tider "pyntede" de på programmet efter sådanne udtalelser, men som regel var der tale om en konstatering, som ikke betød noget særligt for den fortsatte opgaveløsning.

Tilsyneladende er det at skrive pæne programmer en slags, ideal som man efter forsøgspersonernes mening bør stile efter, men som man i praksis ofte går på kompromis med. Ofte kræver den pæne løsning, at man tænker lidt længere frem og overskuer hele løsningen på forhånd, og som nævnt var det ikke forsøgspersonernes generelle strategi. Ofte kan det tage lidt længere tid at lave den pænere, generelle løsning, og derfor kan man fristes til at følge den mindre pæne løsning, der falder en ind først. Igen må vi tage forbehold for opgavens ringe størrelse og forsøgspersonernes situation: Det er højst tænkeligt, at de i en mindre presset situation ville have nået frem til en pænere løsning – på den anden side skulle man tro, at vores tilstedeværelse på en måde opfordrede til at ønske at levere så pænt et resultat som muligt.

6. Konklusion

Vi har set en række interessante fænomener i spil hos forsøgspersonerne – de er alle velkendte i en eller anden forstand, men vi mener alligevel, at man kan lære noget af vores resultater.

Vores forsøgspersoner har med al tydelighed demonstreret, at det er lettere at skrive et program, hvis man på forhånd har gjort sig klart, hvad det skal gøre. Man siger ofte, at programmer normalt er så komplekse, at det ikke kan lade sig gøre at overskue hele problemet fra starten. En hjælp til at løse dette problem kommer i form at begrebet "pæn programmering": Hvis man bygger sit program pænt op, vil det være lettere senere at tage højde for evt. nye opdagelser. Vores forsøgspersoner talte meget om, hvor pæn eller grim deres løsning var, men for de flestes vedkommende var der primært tale om, at de fandt deres løsning grim, men ikke ændrede den af den grund.

To faktorer, der kan virke som en hindring, når man skal udvikle programmer, er fiksering og den manglende evne til at lede efter negative eksempler. Det er svært, umiddelbart at komme med gode retningslinier til at undgå disse problemer, men alene det, at være opmærksom på, at problemerne findes, vil formentlig hjælpe. En programmør kan sikkert ofte spare sig selv for at lave fejl, hvis han er mere grundig med at lede efter eksempler, der kan afkræfte hans hypoteser. Tilsvarende kan grundig analyse af problemerne måske afhjælpe fikseringsproblemer.

---

[6] Man kan sige, at strukturen i et program er opdelingen i delproblemer. En fornuftig og overskuelig måde at opdele programmet på giver et velstruktureret program.

Det sidste fænomen, vi har beskæftiget os med, er den mentale model, programmøren opbygger undervejs i arbejdet. På baggrund af vores iagttagelser af, hvilken rolle denne mentale model spiller, ser det ud til at være en fordel at være opmærksom på at støtte sin mentale model, når man skriver programmer. Den gængse måde at gøre dette på er ved opstille programteksten på en måde, så den visuelt fremtræder på samme måde, som den mentale model er organiseret. Valg af fornuftige navn til variable osv. kan også være en hjælp her.

En anden konsekvens af den måde, programmører bruger deres mentale model, er at den til tider kan stille sig i vejen – hvis den mentale model ikke svarer til programmet (f.eks. på grund af en forkert antagelse eller en gemen skrivefejl) kan det være svært at finde fejl, fordi programmøren tager udgangspunkt i sin mentale model i stedet for i selve programmet.

7. Perspektivering

Vi har undersøgt, hvordan fire datalogistuderende løste en opgave, vi stillede dem. Opgaven og omstændighederne omkring den adskiller sig på mange punkter fra udviklingsprojekter i "den virkelige verden". De væsentligste er:

1) Størrelsen af opgaven. Vores forsøgspersoner kunne løse opgaven på ca. 1 time, hvor rigtige udviklingsprojekter gerne strækker sig over flere år. Vores opgave var så lille, at forsøgspersonerne uden problemer kunne overskue hele opgaven i alle detaljer. I større projekter er udvikleren nød til at støtte sig til dokumentationen og til en overordnet, udetaljeret forståelse af hele opgaven, da der kun kan være kognitive ressourcer til at have den fuldt detaljerede forståelse af en lille bid af projektet af gangen.

2) Størrelsen af udviklingsgruppen. Normalt foregår udvikling i projektgrupper med flere medlemmer. Dels vil det tage alt for lang tid, hvis en mand alene skal løse opgaven, og dels har udviklere hvert deres speciale, så det kan være nødvendigt med flere udviklere på en opgave for at få dækket alle områder. Jo flere der er i en udviklingsgruppe, desto større er behovet for kommunikation, så alle i gruppen har den samme forståelse af projektet, og behovet for struktur i udviklingsprocessen, så ingen 'træder hinanden over tæerne' eller laver dobbelt arbejde.

3) Problemdomænet. Domænet for vores opgave var velkendt for alle forsøgspersonerne. Måske var deres viden ikke så klar, at de umiddelbart kunne formulere reglerne for, hvornår der skal være parenteser, men de var alle på forhånd i stand til at sætte parenteser rigtigt i et eksempel. I virkelige opgaver vil der oftest være behov for at udvikleren tilegner sig store mængder viden fra problemdomænet, så han i første omgang er i stand til at forstå kravspecifikationen og senere rette den, hvor den strider mod sig selv eller er uhensigtsmæssig.

4) Arbejdssituationen. Både det at skulle tænke højt og det, at der sad tre personer og 'kiggede over nakken', har nok påvirket forsøgspersonernes måde at arbejde på i forhold til, hvad de ville gøre, hvis de bare sad for sig selv og arbejdede.

5) Forsøgspersonerne. Vores forsøgspersoner kan ikke beskrives som et tilfældigt, repræsentativt udpluk af befolkningen – ikke en gang af mængden af udviklere, så det kan være farligt at forsøge at generalisere ud fra, hvad vi har observeret med dem. De er alle hankøn, 20-25 år, har alle studeret datalogi, bor alle i København osv. Vi spurgte disse fire, da de er en del af vores vennekreds, hvilket også betyder, at de repræsenterer en afgrænset gruppe.

Vores opgave adskiller sig altså på mange væsentlige punkter fra et programmeringsprojekt fra 'den virkelige verden'. Alligevel mener vi, at vores konklusioner kan bruges til noget af udviklere, da vores opgave på mange punkter svarer til en af de mindre delopgaver, som en programmeringsopgave nedbrydes i.

Vores resultater stemmer godt over ens med, hvad andre har set, når de har lavet tilsvarende undersøgelser. Pennington og Grabowski (1990) skriver bl.a. om, hvorledes implementationsarbejde og design er processer, som udviklere hele tiden skifter imellem, og om den 'mentale model' af et program, de mener udvikleren benytter til at overskue effekten af det, han skriver.

8. Referencer

HOC, J.-M., T.R.G GREEN, R. SAMURÇAY, D.J. GILMORE (EDS.) (1990), *Psychology of Programming*, European Association of Cognitive Ergonomics, Academic Press.

PENNINGTON, N, B. GRABOWSKI (1990), *The Tasks of Programming*, i Hoc m.fl. (eds.), Psychology of Programming, Academic Press.

SACERDOTI, E.D. (1977), *A Structure for Plans and Behaviour*, MIT Press.

SMYTH, MARY M., ALAN F. COLLINS, PETER E. MORRIS, PHILIP LEVY (1994), *Cognition in action*, 2nd Edition. Lawrence Erlbaum Associates.

SPELLING, K. (1972), *Intelligens og Tænkning*, Berlingske Forlag.

Appendiks: Forskellige notationer for udtryk

Dette appendiks forklarer forskellige måder at betragte udtryk – den gængse måde, som dataloger kalder infix notation, samt et par andre metoder, som kan være nyttige for den opgave, vi har stillet vores forsøgspersoner.

Almindelig notation

I et (simpelt) matematisk udtryk kan der indgå flere forskellige elementer. Det er tal, operatorer (+, - , *, /).

Et matematisk udtryk kan skrives op på flere forskellige måder. Normalt anvender man infixnotation. Det vil sige at man skriver operatorerne mellem tallene.

1+2 2-3*4 4-2*4+2+9-5*8

Ovenstående er eksempler på infixudtryk. Hver operator svarer til en beregning, der skal foretages for at finde udtrykkes *værdi*. For at afgøre i hvilken rækkefølge man foretager disse beregninger kan man indføre parenteser i udtrykkene. Således er der forskel på de to nedenstående udtryk.

4-2*4+2+9-5*8 4-2*(4+2+(9-5))*8

For man ikke skal skrive en masse parenteser op, benytter man at operatorerne har en præcedens. Det vil sige at man anvender nogle operatorer før andre. Således har * og / højere præcedens end - og +. Det vil sige at i 2-3*4 skal man først regne 3*4 ud og derefter trække det fra 2.

Hvis ovenstående regler ikke er udtømmende tager man operatorerne fra venstre mod højre. Denne sidste 'alt omfattende regel' giver anledning til, at en del parenteser er overflødige.

1+2+3 (1+2)+3

I det første udtryk kan vi således bruge venstre-mod-højre regelen til at afgøre, at 1+2 skal udregnes først og at vi derefter skal lægge 3 til. I det andet tilfælde kan vi bruge parentesreglen til at afgøre, at det er 1+2 der skal lægges sammen først. Vi siger at venstre-mod-højre reglen overflødiggør en parenteserne i det sidste tilfælde.

1+2+3 1+(2+3)

Ovenstående to eksempler har den samme værdi, men de udregnes ikke på samme måde. Det første eksempel udregnes ligesom før, men det andet der regner man summen af 2+3 ud *før* man lægger summen til 1. Man siger at de to udtryk er ens, da de har samme værdi, men de er ikke identiske for de beregnes ikke på samme måde.

Det er imidlertid ikke alle udtryk på samme form som ovenstående der er ens. Om de er ens afhænger af om den anden operator (den inden i parentesen) er associativ. At en operator er associativ vil sige, at det er ligegyldigt om operanderne[7] står på hver sin side af operatoren.

1+2 = 3 2+1 = 3 3-4 = -1 4-3 = 1

Som det ses af ovenstående tilfælde er + en associativ operator, mens - ikke er det. På samme måde er * associativ, mens / ikke er det.

[7] Det en operator virker på, kaldes operatorens operander.

Postfixnotation

At det er så vanskeligt at beskrive regne og parentesreglerne for infixnotation leder en til at finde en anden måde at anskue et matematisk udtryk på. Det er faktisk ikke særlig svært at finde en bedre måde.

1+2 1 2 +

Ovenfor kan man først se et udtryk i infix notation og derefter det tilsvarende udtryk i postfixnotation. Ideen i postfixnotation er, at man skriver tingene op i den rækkefølge man beregner værdien i. 1 2 + skal således læses fra venstre mod højre og det lyder. 1; gør ikke noget ved det, 2; gør ikke noget ved det, +; læg de to foregående sammen (det bliver 3) og gør ikke noget ved det.

4 2 + 1 - → 6 3 - → 5

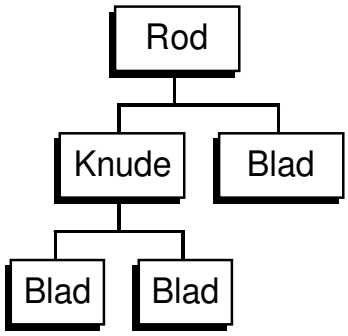
Ovenfor er et eksempel på hvorledes værdien af 4 2 + 1 - regnes ud. Hvis vi ser på eksemplerne i infixnotation, ser man at de steder hvor der var dobbelttydig notation før, er der det ikke længere.

Infixnotation	Postfixnotation	Værdi
1+2+3	1 2 + 3 +	6
1+(2+3)	1 2 3 + +	6
4-3	4 3 -	1
3-4	3 4 -	-1
3-2-1	3 2 - 1 -	0
3-(2-1)	3 2 1 - -	2

Som det ses af ovenstående løser postfixnotationen alle tvivlsspørgsmål, der skal have regler om præcedens og associativitet i infixnotationen.

Udtrykstræer

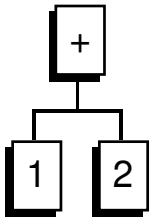
Et træ er en måde at betragte data på. I et træ indgår knuder, grene, blade og roden som elementer. I eksemplet **Fejl! Ukendt argument for parameter.**, ses et træ med en rod, tre blade og en knude. Her er roden også at betragte som en knude. Man siger at en knude har en venstre og et højre barn. Således har roden to børn. Det venstre barn er igen en knude, mens det højre barn er et blad. Ofte omtales et barn med alle sine 'efterkommere' som et deltræ eller et undertræ.



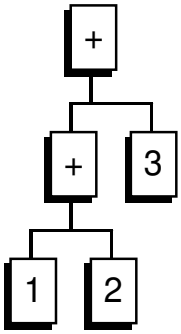
Figur Fejl! Ukendt argument for parameter. **: Eksempel på et træ**

Et træ kan bruges til at beskrive et matematisk udtryk med. Hvis man i knuderne sætter operatorer kan man betragte en knudes børn, som operatorens operander. Når man vil beregne værdien af et udtryk beregner man det venstreste deltræ først; er der ikke noget at beregne der, beregner man værdien af det højre deltræ og det bliver man ved med, til man kun har en værdi tilbage i roden.

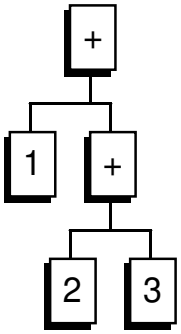
Her i **Fejl! Ukendt argument for parameter.** har vi et eksempel på hvorledes et udtrykstræ kan skrives op. I **Fejl! Ukendt argument for parameter.** er der et lidt mere avanceret tilfælde og endelig i **Fejl! Ukendt argument for parameter.** kan man se, hvorledes en anderledes placering af parenteserne påvirker



Figur Fejl! Ukendt argument for parameter.**: 1+2 som udtrykstræ**

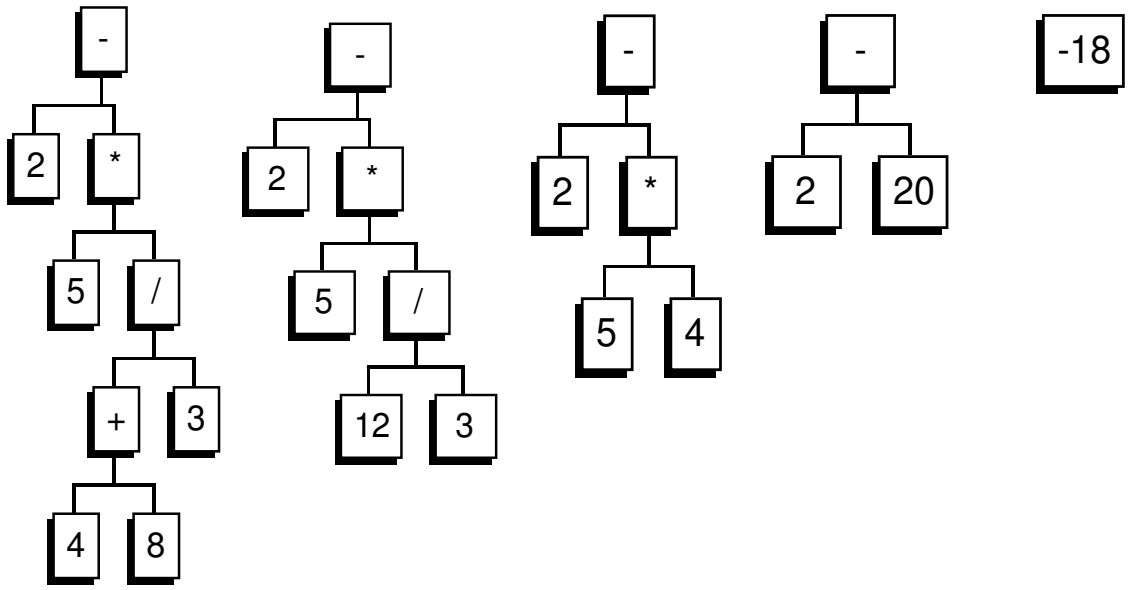


Figur Fejl! Ukendt argument for parameter.**: 1+2+3**



Figur Fejl! Ukendt argument for parameter.**: 1+(2+3)**

hvorledes træet kommer til at se ud. Her under kommer et eksempel på hvorledes man kan regne med træer.



Vi starter med infixudtrykket 2-5*((4+8)/3) og regner det om til værdien -18. Bemærk, at der i infixudtrykket ikke indgår nogle overflødige parenteser, men at man godt kan sætte nogle flere (f.eks. 2-((5)*((4+8)/(3)))).