Indhold

1 Introduktion

Rasmus er et databasesystem, der er baseret på den såkaldte *relationsalgebra*. Det understøtter desuden den *funktionelle* programmeringsstil.

1.1 Relationsalgebra

En *relation* er en tabel som den følgende

Årskort:Int	Navn:Text	Linje:Text
960001	Trine Sørensen	dat/mat
960002	Kurt Petersen	dat/fys
960003	Knud Børge Jensen	mat/dat

Overskriften på relationen kaldes for dens *skema*. Det indeholder et navn for hver søjle, kaldet en *attribut*, sammen med en *type*, der beskriver det lovlige indhold af felterne i søjlen. I eksemplet er benyttet typen `Int`, der omfatter heltallene, og typen `Text`, der omfatter tegnfølger. Hver række i relationen kaldes et *tupel*. For hver attribut rummer det en værdi af den pågældende type.

Man følger den konvention, at rækkefølgen af søjlerne ikke er signifikant. Tilsvarende kan man bytte rundt på tuplerne uden at ændre ved relationen. Eksemplet ovenfor kan altså ligesåvel skrives som

Navn:Text	Årskort:Int	Linje:Text
Knud Børge Jensen	960003	mat/dat
Trine Sørensen	960001	dat/mat
Kurt Petersen	960002	dat/fys

Det kræves altid, at der ikke er to ens rækker i en relation; den kan altså opfattes som en *mængde* af tupler, hvor hvert tupel er en *funktion* fra attributnavne til værdier.

Gennem Rasmus systemet kan man skabe, modificere og inspicere relationer. De praktiske aspekter af dette er i detaljer beskrevet i en separat brugervejledning.

Det er vigtigt, at man kan *kombinere* sine oplysninger. Det sker i Rasmus gennem den såkaldte *relationsalgebra*, hvor man anvender *operatorer* på relationer. En operator tager som operander et antal relationer – og måske andre værdier

– og opbygger på systematisk vis en ny relation. Der er en traditionel samling af sådanne operatorer, der vil blive præsenteret i kapitel 3.

En database skal indholde mere end blot *data* i form af relationer. Man skal kunne definere en samling *forespørgsler*, der tager udgangspunkt i de tilgængelige data. Til dette formål indeholder RASMUS et funktionsbegreb, der tillader, at man definerer parametriserede relationsudtryk. Man har faktisk den fulde frihed, der kendes fra funktionelle programmeringssprog. Dette aspekt præsenteres i kapitel 4.

RASMUS understøtter således flere slags værdier: relationer og funktioner, men også tupler og atomare værdier er til stede. I kapitel 5 beskrives et antal forskellige operatorer på disse. Deres tilstedeværelse vil også gøre de relationelle operatorer mere slagkraftige.

Gennemgangen af RASMUS systemet er hovedsagelig drevet af eksempler. I kapitel 6 diskuteres de generelle begreber, specielt som de relaterer sig til øvrige programmeringssprog.

1.2 Fodboldmaterialet

Som eksempel på en database er valgt en del af resultaterne fra den første sæson i den danske superliga i fodbold, som blev spillet i foråret 1991. For at begrænse størrelsen af relationerne i eksemplerne er turneringen skåret ned til de seks hold, der endte med at være bedst placerede: Brøndby, Lyngby, AGF, Frem, OB og AaB. Det er deres indbyrdes kampe i en dobbeltturnering, der udgør materialet.

Udover informationen om kampene og deres resultater indeholder databasen også informationer om en ugentlig "tipskupon", som altså – igen af pladshensyn – kun indeholder tre kampe.

Informationen er opdelt på relationer med følgende navne og skemaer:

Runde1:

Id:Int	HjHold:Text	UdeHold:Text

Runde2:

Id:Int	HjHold:Text	UdeHold:Text

Resultater:

Id:Int	HjScore:Int	UdeScore:Int

Tips:

Id:Int	Uge:Int	Nr:Int

Spilledag:

Kamp:Int	Dato:Text

De to første relationer indeholder en angivelse af de spillede kampe; der er en relation for hver runde i dobbeltturneringen. Den tredje relation indeholder resultaterne af samtlige kampe; den fjerde angiver uge for uge udseendet af tipskuponerne; den sidste relation angiver, på hvilke dage de enkelte kampe blev spillet. Indholdet af relationerne er

Runde1:

Id:Int	HjHold:Text	UdeHold:Text
1	AGF	Brøndby
2	Frem	OB
3	AaB	Lyngby
4	OB	AGF
5	Brøndby	AaB
6	Lyngby	Frem
7	AaB	AGF
8	Frem	Brøndby
9	Lyngby	OB
10	AGF	Frem
11	Brøndby	Lyngby
12	OB	AaB
13	Frem	AaB
14	Lyngby	AGF
15	OB	Brøndby

Runde2:

Id:Int	HjHold:Text	UdeHold:Text
16	AaB	Frem
17	AGF	Lyngby
18	Brøndby	OB
19	Brøndby	AGF
20	OB	Frem
21	AGF	OB
22	Lyngby	AaB
23	Frem	AGF
24	AaB	Brøndby
25	Frem	Lyngby
26	AGF	AaB
27	Brøndby	Frem
28	OB	Lyngby
29	Lyngby	Brøndby
30	AaB	OB

Resultater:

Id:Int	HjScore:Int	UdeScore:Int
1	1	2
2	3	0
3	1	2
4	0	0
5	2	2
6	1	1
7	4	1
8	1	1
9	1	0
10	2	1
11	0	3
12	1	1
13	1	2
14	2	1
15	1	1
16	1	2
17	1	1
18	0	0
19	0	0
20	1	1
21	3	1
22	2	0
23	2	2
24	0	2
25	2	1
26	6	1
27	4	1
28	1	1
29	1	1
30	5	1

Tips:

Id:Int	Uge:Int	Nr:Int
1	12	2
2	12	3
3	12	1
4	14	3
5	14	1
6	14	2
7	16	1
8	16	2
9	16	3
10	18	1
11	18	2
12	18	3
13	19	1
14	19	2
15	19	3

Spilledag:

Kamp:Int	Dato:Text
1	316
2	317
3	401
4	324
5	407
6	407
7	414
8	414
9	428
10	421
11	505
12	505
13	512
14	512
15	512
16	516
17	516
18	516
19	520
20	520
21	523
22	526
23	530
24	602
25	602
26	609
27	609
28	619
29	623
30	623

2 Rasmus systemet

RASMUS systemet, der benyttes gennem EMACS, er beskrevet i en speciel brugervejledning. Når det startes, så ser man to buffere. I den ene skrives *udtryk*, hvis resultater vises i den anden. Man kan altid navngive og gemme det sidste resultat. I denne note vil vi med notationen `Navn` angive, at det sidste resultat er gemt under navnet `Navn`.

3 Relationsudtryk

Udover at skabe og udskrive relationer, kan man naturligvis også kombinere relationer med hinanden. Dette foregår efter *relationsalgebraens* principper, og den letteste måde at forklare dette på er via en analogi med de sædvanlige regningsarter på heltal. Et aritmetisk heltalsudtryk som

$$7 + 5$$

betyder, at operatoren + skal anvendes på operanderne 7 og 5. Resultatet heraf (udtrykkets værdi) er som bekendt 12, og en sådan værdi kan man så gøre et eller andet videre ved, fx kan man gemme den.

I RASMUS kan relationer optræde i relationsudtryk på samme måde, som tal kan optræde i aritmetiske udtryk. Operatorerne er nogle andre, men betydningen er i princippet den samme: resultatet af at anvende en relationsoperator er en relation – præcis ligesom resultatet af at anvende en heltalsoperator er et heltal.

Syntaksen for relationsoperatorer i RASMUS er nogenlunde den samme som for de sædvanlige regningsarter, idet vi også benytter fx + og $*$, men nu med en anden betydning.

3.1 Union

Man kan danne en relation, som er en foreningsmængde af to andre relationer. Dette sker ved hjælp af den såkaldte *union*-operator, som vi udtrykker ved hjælp af **+** på denne måde

relation$_1$ + relation$_2$

Det kræves, at de to operander har samme skema. Udtrykkets resultat er en ny relation, som man kan bruge overalt, hvor en relation ellers er tilladt, fx som operand til en ny relationsoperator.

Eksempel 1

Skab en relation bestående af samtlige tupler fra de to turneringsrunder

```
Runde1 + Runde2
```

Hvis vi gemmer resultatet af dette udtryk under navnet `Kampe` vil relationen `Kampe` have følgende indhold

Id:Int	HjHold:Text	UdeHold:Text
1	AGF	Brøndby
2	Frem	OB
3	AaB	Lyngby
4	OB	AGF
5	Brøndby	AaB
6	Lyngby	Frem
7	AaB	AGF
8	Frem	Brøndby
9	Lyngby	OB
10	AGF	Frem
11	Brøndby	Lyngby
12	OB	AaB
13	Frem	AaB
14	Lyngby	AGF
15	OB	Brøndby
16	AaB	Frem
17	AGF	Lyngby
18	Brøndby	OB
19	Brøndby	AGF
20	OB	Frem
21	AGF	OB
22	Lyngby	AaB
23	Frem	AGF
24	AaB	Brøndby
25	Frem	Lyngby
26	AGF	AaB
27	Brøndby	Frem
28	OB	Lyngby
29	Lyngby	Brøndby
30	AaB	OB

Union-operatorens måde at kombinere relationer på kan siges at være "lodret", idet resultatrelationen bliver længere (i alt fald ikke kortere) end den længste af operanderne.

3.2 Join

Man kan også kombinere relationer "vandret", det vil sige, så resultatet bliver bredere (i alt fald ikke smallere) end den bredeste af operanderne. Operatoren hertil hedder *join* og optræder i udtryk i form af `*` på følgende måde

$$\textbf{relation}_1 \texttt{ * } \textbf{relation}_2$$

Resultatet af dette udtryk er en ny relation, hvis attributnavne udgøres af alle de attributnavne, der findes i de to operander tilsammen. Eventuelle attributnavne, der optræder begge steder, kommer dog kun med en enkelt gang; det kræves, at de to operander er enige om typerne for sådanne fælles attributnavne.

Tuplerne i den nye relation dannes ved at kombinere alle par af tupler fra de to relationer, som har samme værdier på attributter med samme navn.

Join-operatoren kan give en tom relation som resultat, hvis der ikke er to tupler med samme værdi på de fælles attributter. Hvis de to operander ikke har nogen fælles attributter, vil resultatet indeholde alle mulige kombinationer af deres tupler.

Eksempel 2

Skab en relation, der indeholder samtlige kampe og deres resultater

```
Kampe * Resultater
```

Hvis resultatet af udtrykket gemmes under navnet `Turnering` er værdien af `Turnering` som følger

Id:Int	HjHold:Text	UdeHold:Text	HjScore:Int	UdeScore:Int
1	AGF	Brøndby	1	2
2	Frem	OB	3	0
3	AaB	Lyngby	1	2
4	OB	AGF	0	0
5	Brøndby	AaB	2	2
6	Lyngby	Frem	1	1
7	AaB	AGF	4	1
8	Frem	Brøndby	1	1
9	Lyngby	OB	1	0
10	AGF	Frem	2	1
11	Brøndby	Lyngby	0	3
12	OB	AaB	1	1
13	Frem	AaB	1	2
14	Lyngby	AGF	2	1
15	OB	Brøndby	1	1
16	AaB	Frem	1	2
17	AGF	Lyngby	1	1
18	Brøndby	OB	0	0
19	Brøndby	AGF	0	0
20	OB	Frem	1	1
21	AGF	OB	3	1
22	Lyngby	AaB	2	0
23	Frem	AGF	2	2
24	AaB	Brøndby	0	2
25	Frem	Lyngby	2	1
26	AGF	AaB	6	1
27	Brøndby	Frem	4	1
28	OB	Lyngby	1	1
29	Lyngby	Brøndby	1	1
30	AaB	OB	5	1

3.3 Select

Det er ovenfor vist, at relationer kan gøres længere og bredere, så det er ikke særligt overraskende, at de også kan gøres kortere og smallere.

En måde at gøre dem kortere på er at udvælge de tupler, der opfylder en given betingelse. Hertil anvendes den såkaldte *select*-operator, som vi skriver på følgende måde

relation ? betingelse

Resultatet er en relation, som indeholder netop de tupler, der opfylder betingelsen.

Eksempel 3

Skab en relation bestående af de kampe i turneringen, hvor hjemmeholdet vandt

```
Turnering ? (#.HjScore > #.UdeScore)
```

Værdien af dette udtryk er

Id:Int	HjHold:Text	UdeHold:Text	HjScore:Int	UdeScore:Int
2	Frem	OB	3	0
7	AaB	AGF	4	1
9	Lyngby	OB	1	0
10	AGF	Frem	2	1
14	Lyngby	AGF	2	1
21	AGF	OB	3	1
22	Lyngby	AaB	2	0
25	Frem	Lyngby	2	1
26	AGF	AaB	6	1
27	Brøndby	Frem	4	1
30	AaB	OB	5	1

■

Betingelsen `#.HjScore > #.UdeScore` er et eksempel på en *simpel* betingelse. Vi skal senere se, at man kan skrive langt mere avancerede betingelser i RASMUS, og vi vil også forklare hvad `#` betyder. For nærværende er det tilstrækkeligt at forstå, at ovennævnte betingelse udvælger de tupler, hvor værdien af attributten `HjScore` er større end værdien af attributten `UdeScore`.

3.4 Project

Hvis relationer skal gøres smallere, anvendes den såkaldte *project*-operator, som vi skriver som

relation |+ attributliste

Resultatet er den del af operandrelationen, der udgøres af attributterne i den angivne attributliste, medens de øvrige attributter "smides væk". Bemærk, at project-operatoren samtidigt kan gøre relationen kortere, da to tupler fra operanden kan blive ens, når nogle af attributterne fjernes.

Eksempel 4

Skab en relation, der for hver kamp indeholder det antal mål, der er scoret af udeholdet

```
Resultater |+ Id,UdeScore
```

Værdien af udtrykket er

Id:Int	UdeScore:Int
1	2
2	0
3	2
4	0
5	2
6	1
7	1
8	1
9	0
10	1
11	3
12	1
13	2
14	1
15	1
16	2
17	1
18	0
19	0
20	1
21	1
22	0
23	2
24	2
25	1
26	1
27	1
28	1
29	1
30	1

■

Vi kunne have opnået samme resultat ved at bruge det alternative udtryk

```
Resultater |- HjScore
```

idet man ved varianten `|-` angiver de attributter, der skal fjernes.

3.5 Sammensatte udtryk

Alle relationsudtryk i de foregående eksempler har været "flade", i den forstand, at operanderne til de forskellige relationsoperatorer har været simple relationer. Operander kan imidlertid selv være udtryk, hvis operander igen kan være udtryk, og så videre.

På samme måde som man i et aritmetisk udtryk kan skrive fx

$$(7 + 5) * 3$$

kan man i relationsalgebraen skrive fx

```
(Runde1 + Runde2) |+ Id
```

Dette sammensatte udtryk beregner identiteterne på samtlige kampe i dobbeltturneringen.

3.6 Rename

Foruden join og project er der endnu en operator, som kan siges at virke "vandret". Det er den såkaldte *rename*-operator, der er nødvendig, hvis en relations attributnavne ikke er hensigtsmæssige for en efterfølgende brug. Antag fx, at der ønskes foretaget en kombination af relationerne `Runde1` og `Spilledag`, således at kampe, der optræder i begge relationer, bliver repræsenteret ved de samlede oplysninger om dem. Man kan ikke umiddelbart benytte join, da identitetsattributterne har forskellige navne; i `Runde1` hedder den `Id`, og i `Spilledag` hedder den `Kamp`. Hvis vi join'ede de to relationer ville resultatet blive en relation med 450 tupler, idet alle tupler i `Runde1` ville blive parret med alle tupler i `Spilledag`. Dette er ikke hvad vi ønsker, og det er derfor nødvendigt at kunne omdøbe attributnavne. Det gøres ved hjælp af *rename*-operatoren, som anvendes på følgende måde

relation `[` navneparliste `]`

I **navneparliste** står de udskiftninger af navne, som vi ønsker at foretage, på formen **gammelt navn `<-` nyt navn**, adskilte af kommaer.

Eksempel 5

Værdien af udtrykket `Spilledag [ Kamp <- Id ]` er som følger

Id:Int	Dato:Text
1	316
2	317
3	401
4	324
5	407
6	407
7	414
8	414
9	428
10	421
11	505
12	505
13	512
14	512
15	512
16	516
17	516
18	516
19	520
20	520
21	523
22	526
23	530
24	602
25	602
26	609
27	609
28	619
29	623
30	623

Nu kan vi udtrykke nyttige kombinationer af `Runde1` og `Spilledag`.

Eksempel 6

Skab en relation bestående af datoerne for AGF's hjemmekampe i første runde af turneringen

```
((Runde1 ? (#.HjHold="AGF"))*(Spilledag[Kamp<-Id])) |+ Dato
```

Værdien er

Dato:Text
316
421

3.7 Difference

Vi har også en såkaldt *difference*-operator, som bruges til at finde de tupler, der optræder i een relation, men ikke i en anden. Udseendet er

$$\textbf{relation}_1 \texttt{ - } \textbf{relation}_2$$

Det kræves som ved union, at de to operander har samme skema.

Eksempel 7

Find ud af, hvilke hold der ikke har vundet på udebane

```
(Turnering |+ UdeHold) -
((Turnering ? (#.HjScore<#.UdeScore)) |+ UdeHold)
```

Værdien af udtrykket er

UdeHold:Text
AGF
OB

Bemærk, at det her er vigtigt, at relationer ikke indeholder dubletter. Værdien af det første deludtryk ovenfor, `Turnering |+ UdeHold`, er

UdeHold:Text
AGF
Frem
AaB
OB
Brøndby
Lyngby

Vi ser, at hvert hold kun optræder en enkelt gang, uanset hvor mange kampe, det har spillet. Dette skyldes som tidligere nævnt, at relationer er *mængder*, og dette er en vigtig forudsætning for, at difference-operatoren giver mening.

Når RASMUS således er udstyret med de to mængde-operatorer union og difference for foreningsmængde og mængdedifferens, så ville det være at forvente, at der også var en operator for fællesmængde. Det er der også – men kun indirekte – fordi det er nemt at se, at hvis to relationer har samme skema, så vil join beregne deres fællesmængde. Resultatet vil indeholde de tupler, der har samme værdi på alle attributter, og det er jo netop de fælles tupler.

3.8 Indskudte udtryk

Vi slutter dette kapitel med et lidt mere kompliceret eksempel, hvor vi bruger `Tips`-relationen for første gang.

Eksempel 8

Find nummeret på den kamp på tipskuponen for uge 18, hvor AGF var med

```
(((Kampe*(Tips ? (#.Uge=18))) ? (#.UdeHold="AGF")) +
 ((Kampe*(Tips ? (#.Uge=18))) ? (#.HjHold="AGF"))) |+ Nr
```

Resultatet bliver

Nr:Int
1

∎

Som vi skal se i et senere kapitel, kan dette udtryk simplificeres en smule, men alligevel kan vi allerede nu se, at når relationsudtryk bliver store, så bliver der behov for at kunne opskrive dem på en mere hensigtsmæssig måde, end vi har set hidtil. Til den ende har RASMUS følgende midlertidige navngivningsmekanisme, et såkaldt *indskudt udtryk*

```
(+  Val id1 = relation1
    Val id2 = relation2
    ⋮
    Val idn = relationn
    in relation
+)
```

Dette udtryk bevirker, at man under udregningen af det sidste udtryk **relation** kan referere til værdierne af $\mathbf{relation_1}$ til $\mathbf{relation_n}$ ved hjælp af navnene $\mathbf{id_1}$, ..., $\mathbf{id_n}$. Resultatet af det samlede `(+ ... +)`- udtryk er den således udregnede værdi af **relation**. Navnene $\mathbf{id_1}$, ..., $\mathbf{id_n}$ er *lokale* og kan ikke refereres uden for `(+ ... +)`- udtrykket. I udtrykket $\mathbf{relation_i}$ kan man benytte navnene $\mathbf{id_1}$, ..., $\mathbf{id_{i-1}}$.

Eksempel 9

Udtrykket i eksempel 8 kan også skrives på følgende facon

```
(+  Val X = Kampe*(Tips ? (#.Uge=18))
    in ((X ? (#.HjHold="AGF"))+(X ? (#.UdeHold="AGF"))) |+ Nr
+)
```

4 Funktioner

RASMUS er også udstyret med en *funktionsmekanisme*, der dels kan bruges til at *parametrisere* relationsudtryk og dels til at opbygge *biblioteker* af nyttige udtryk.

Betragt igen eksempel 8. Det er sandsynligt, at hvis man er interesseret i nummeret på AGF's tipskamp i uge 18, så er man formentlig også interesseret i numrene på andre holds tipskampe i andre uger.

Eksempel 10

Skriv en funktion, der, givet et ugenummer og navnet på et hold, returnerer en relation, der indeholder holdets kampnummer på den pågældende tipskupon

```
Func (uge:Int, hold:Text) -> (Rel)
  (+ Val X = Kampe*(Tips ? (#.Uge=uge))
     in ((X ? (#.HjHold=hold))+(X ? (#.UdeHold=hold))) |+ Nr
  +)
end
```

Hvis værdien af dette udtryk gemmes under navnet `FindNr` vil følgende anvendelse af funktionen

```
FindNr(18,"AGF")
```

give samme resultat som udtrykkene i eksempel 8 og 9. Udtrykket

```
Func (uge:Int, hold:Text) -> (Rel)
  ...
end
```

er en såkaldt *funktionsdefinition*, medens udtrykket

```
FindNr(18,"AGF")
```

er en *funktionsanvendelse*. Funktionens *hoved*

```
Func (uge:Int, hold:Text) -> (Rel)
```

angiver at der er tale om en funktion med to *parametre*, henholdsvis `uge` og `hold`, hvis *typer* er henholdsvis `Int` og `Text`; man kan desuden se, at funktionens resultat er af type `Rel`. Navnene `uge` og `hold` er *formelle* parametre.

Funktionen er beregnet til at blive anvendt med forskellige *aktuelle* parametre; i eksemplet ovenfor er `18` og `"AGF"` således aktuelle parametre. Man kan naturligvis også bruge andre aktuelle parametre, som i anvendelsen

```
FindNr(19,"Lyngby")
```

Betydningen af en funktionsanvendelse er, at det udtryk, der udgør funktionens *krop* – her udtrykket `(+ ... +)` – beregnes med de formelle parametre erstattet med værdierne af de aktuelle parametre.

Typenavnene `Int` og `Text` angiver, at den første aktuelle parameter skal være af typen `Int`, det vil sige et heltal, medens den anden skal være af typen `Text`, som betegner mængden af tegnfølger. Typeangivelsen `(Rel)` på højresiden af `->` fortæller, at resultatet er af type `Rel`, som betegner mængden af relationer.

I de næste to eksempler definerer vi to funktioner, der er nyttige, hvis vi skal konstruere en tipskupon for en given uge. Vi antager, at vi først har anvendt RASMUS systemet til at konstruere følgende tre relationer:

Et:

Tegn
1

Kryds:

Tegn
X

To:

Tegn
2

Eksempel 11

Skriv en funktion, der for en given uge konstruerer relationen

HjHold:Text	UdeHold:Text	HjScore:Int	UdeScore:Int	Nr:Int

bestående af kampene på den pågældende uges tipskupon

```
Func (uge:Int) -> (Rel)
  ((Turnering*Tips) ? (#.Uge=uge)) |- Id,Uge
end
```

Hvis denne funktion gemmes under navnet `Strip`, så kan vi benytte den til at løse problemet.

Eksempel 12

Skriv en funktion, der for en given uge konstruerer tipskuponen

```
Func (ugenr:Int) -> (Rel)
  (((Strip(ugenr) ? (#.HjScore < #.UdeScore))*To) +
   ((Strip(ugenr) ? (#.HjScore = #.UdeScore))*Kryds) +
   ((Strip(ugenr) ? (#.HjScore > #.UdeScore))*Et)
  ) |- HjScore,UdeScore
end
```

■

Hvis denne nu gemmes under navnet `Kupon` har vi det ønskede resultat. Fx vil anvendelsen `Kupon(18)` have værdien

Nr:Int	HjHold:Text	UdeHold:Text	Tegn:Text
2	Brøndby	Lyngby	2
3	OB	AaB	X
1	AGF	Frem	1

5 Generelle udtryk

De udtryk, vi hidtil har beskæftiget os med, har næsten alle været *relationsudtryk*, det vil sige, udtryk hvis resultater er relationer, og hvor de indgående operatorer er relationsoperatorerne union, join og så videre. RASMUS tillader imidlertid udtryk af flere forskellige typer, herunder *boolske* udtryk og *aritmetiske* udtryk, hvis resultater er henholdvis sandhedsværdier og heltal. En betingelse i en select-operator er således i det generelle tilfælde et boolsk udtryk.

5.1 Typer og konstanter

Indtil nu har vi stiftet bekendskab med følgende typer

`Int`:	heltal
`Text`:	tegnfølger
`Rel`:	relationer
`Func`:	funktioner

Der findes endnu to typer i RASMUS

`Bool`:	sandhedsværdier
`Tup`:	tupler

Typerne `Int`, `Text` og `Bool`, hvis værdier er dem der kan stå i de enkelte felter i relationer, kaldes *atomare* og kan under et angives med fællesbetegnelsen `Atom`. Den totale samling af de seks typer kan angives med fællesbetegnelsen `Any`.

De atomare typer har de sædvanlige *konstanter*, det vil sige

```
 Int:  ..., -1, 0, 1, 2, ...
Bool:  true, false
Text:  "dette er en konstant af type Text"
```

At konstanter af type `Text` omgives med `"` og `"` skyldes, at det er nødvendigt at kunne skelne *navnet* `AGF` fra tegnfølgen `"AGF"`, som er tegnet `A` efterfulgt af tegnet `G` efterfulgt af tegnet `F`.

Typen `Tup` har også en slags konstanter, idet et udtryk af formen

$$\texttt{Tup(}\mathbf{attribut_1 : værdi_1}\texttt{, } \ldots \texttt{,}\mathbf{attribut_n : værdi_n}\texttt{)}$$

angiver et tupel med n felter, hvis attributnavne er $\mathbf{attribut_1}, \ldots, \mathbf{attribut_n}$, og hvis tilhørende værdier er $\mathbf{værdi_1}, \ldots, \mathbf{værdi_n}$. Man kan også konstruere en slags konstante relationer; hvis `t` nemlig er et tupel, så er

```
Rel(t)
```

en relation bestående af det enkelte tupel `t`.

Eksempel 13

Skab en relation bestående af de tre tipstegn

```
Rel(Tup(Tegn:"1"))+Rel(Tup(Tegn:"X"))+Rel(Tup(Tegn:"2"))
```

Værdien af dette udtryk er

Tegn:Text
1
X
2

∎

Vi kan nu skrive funktionen `Kupon` i eksempel 12 på følgende alternative måde. Bemærk, at vi kan bruge tidligere `Val`-definitioner i senere `Val`-definitioner.

Eksempel 14

```
Func (ugenr:Int) -> (Rel)
  (+ Val Strip =
         Func (uge:Int) -> (Rel)
           ((Turnering*Tips) ? (#.Uge=uge)) |- Id,Uge
         end
     Val X = Strip(ugenr)
     Val Et =
         (X ? (#.HjScore>#.UdeScore))*Rel(Tup(Tegn:"1"))
     Val Kryds =
         (X ? (#.HjScore=#.UdeScore))*Rel(Tup(Tegn:"X"))
     Val To =
         (X ? (#.HjScore<#.UdeScore))*Rel(Tup(Tegn:"2"))
     in (Et+Kryds+To) |- HjScore,UdeScore
  +)
end
```

5.2 Sammensatte betingelser

Vi illustrerer brugen af boolske udtryk i de følgende tre eksempler. I det første eksempel skriver vi en alternativ udgave af eksempel 8.

Eksempel 15

Find nummeret på den kamp på tipskuponen for uge 18, hvor AGF var med

```
((Kampe*(Tips ? (#.Uge=18))) ?
    ((#.Udehold="AGF") or (#.HjHold="AGF"))) |+ Nr
```

Vi kan få brug for mere komplicerede boolske udtryk.

Eksempel 16

Skab en relation bestående af de kampe, der fik en vinder, og hvor taberen ikke scorede

```
Turnering ? (((#.UdeScore > 0) and (#.HjScore = 0)) or
             ((#.UdeScore = 0) and (#.HjScore > 0)))
```

Værdien af udtrykket er

Id:Int	HjHold:Text	UdeHold:Text	HjScore:Int	UdeScore:Int
2	Frem	OB	3	0
9	Lyngby	OB	1	0
11	Brøndby	Lyngby	0	3
22	Lyngby	AaB	2	0
24	AaB	Brøndby	0	2

■

Man kan også få brug for betingelser, hvor der indgår beregninger på fx tuplernes attributværdier.

Eksempel 17

Find de kampe, hvor hjemmeholdet vandt med mere end to mål

```
Turnering ? (#.HjScore - #.UdeScore > 2)
```

■

5.3 Forall og tupeludtryk

I det følgende skal vi beskrive mere præcist, hvordan et generelt boolsk udtryk ser ud, men først skal vi introducere den såkaldte *forall*-operator, som er en relationsoperator, der giver mulighed for at konstruere relationer ved at "regne på" *enkelttupler* fra andre relationer. Operatoren har følgende udseende

!(relation): udtryk

hvor **udtryk** skal være af type `Rel`. Betydningen af et forall-udtryk er, at tuplerne i **relation** et for et "udsættes" for den mekanisme, der angives af **udtryk**, hvorefter resultaterne heraf forenes til en relation, som så er resultatet af hele forall-udtrykket.

Betragt følgende eksempel, hvor vi ønsker at udregne, hvad vi kan kalde underholdningsværdien af de enkelte fodboldkampe, defineret som antal mål scoret af hjemmeholdet plus to gange antal mål scoret af udeholdet.

Eksempel 18

Udvid relationen `Turnering` med en attribut, som for hver kamp indeholder dennes underholdningsværdi

```
!(Turnering): Rel(# « Tup(UV: #.HjScore+2*#.UdeScore))
```

Værdien af udtrykket er

Id:Int	...	HjScore:Int	UdeScore:Int	UV: Int
6		1	1	3
25		2	1	4
10		2	1	4
23		2	2	6
12		1	1	3
30		5	1	7
11		0	3	6
29		1	1	3
1		1	2	5
19		0	0	0
2		3	0	3
20		1	1	3
13		1	2	5
16		1	2	5
15		1	1	3
18		0	0	0
5		2	2	6
24		0	2	4
8		1	1	3
27		4	1	6
14		2	1	4
17		1	1	3
9		1	0	1
28		1	1	3
4		0	0	0
21		3	1	5
3		1	2	5
22		2	0	2
7		4	1	6
26		6	1	8

Dette eksempel illustrerer flere ting. For det første kan vi se, at udtrykket på højresiden af `:` er af formen

```
Rel(# « t)
```

hvor `t` er et tupel med en enkelt attribut, `UV`. Værdien af denne attribut er

```
#.HjScore+2*#.UdeScore
```

der udregnes på følgende måde. Symbolet `#` er *standardnavnet* på hvad vi kan kalde det *løbende* tupel i forall-udtrykket, og `#.HjScore` angiver derfor værdien

af attributten `HjScore` i dette tupel. Givet `#` er værdien af `#.HjScore+2*#.UdeScore` således lig med underholdningsværdien af `#`. I udtrykket `Rel(# « t)` angiver `t` derfor et tupel med eneste attribut `UV`, hvis værdi er underholdningsværdien af `#`. Udtrykket `# « t` angiver nu resultatet af at anvende *tupeloperatoren* « på de to tupler `#` og `t`. Operatoren « er en *udvidelses- og overskrivningsoperator*, der virker som følger. Resultatet af

```
t1 « t2
```

er det tupel, der fås ved først at kopiere `t1`, dernæst, hvis der er fælles attributter i `t1` og `t2`, at erstatte værdierne i `t1` med de tilsvarende fra `t2`, og endeligt at udvide med de attributter fra `t2`, der ikke findes i `t1`. Følgende eksempel illustrerer dette.

Eksempel 19

Resultatet af tupeludtrykket

```
Tup(Dig:1, Mig:2) « Tup(Mig:3, ViTo:4)
```

er lig med `Tup(Dig:1, Mig:3, ViTo:4)`. ∎

Vi kan nu se, at udtrykket `# « t` ovenfor netop angiver et tupel fra `Turnering` udvidet med den pågældende kamps underholdningsværdi. Derfor angiver hele udtrykket `Rel(# « t)` en *relation* indeholdende netop dette tupel. Resultatet af hele forall-udtrykket i eksempel 18 består af foreningsmængden af alle disse relationer, det vil sige, af samtlige kampe i turneringen udvidet med en underholdningsværdi-attribut.

Der findes endnu en tupeloperator, nemlig *eliminationsoperatoreren* \, der bruges som følger

```
t\A
```

hvor `t` er et tupel og `A` er en attribut. Resultatet er en kopi af tuplet `t`, med en eventuel `A`-attribut fjernet. Vi kan illustrere denne operator ved hjælp af følgende alternative udgave af eksempel 4.

Eksempel 20

Skab en relation, der for hver kamp indeholder det antal mål, der er scoret af udeholdet

```
!(Resultater): Rel(# \ HjScore)
```

Med introduktionen af forall-operatoren og den dertil hørende forklaring på betydningen af `#` skulle det nu være klart, at `#` spiller præcist samme rolle i select-udtryk, som den gør i forall-udtryk. I select-operatoren "gennemløbes" relationens tupler på samme vis, og de tupler, der opfylder betingelsen "smutter forbi", medens de øvrige "filtreres fra".

5.4 Betingede udtryk

RASMUS er også udstyret med *betingede udtryk*, der ser ud som følger

```
if betingelse_1 -> udtryk_1
 & betingelse_2 -> udtryk_2
 ⋮
 & betingelse_n -> udtryk_n
fi
```

Udtrykket kan bruges til at vælge mellem alternativer, som beskrevet af følgende eksempel.

Eksempel 21

Skriv en funktion, der givet ugenummer og navnet på et hold beregner en relation, der fortæller, om det pågældende hold er på ude- eller hjemmebane i den pågældende uge

```
Func (uge:Int, hold:Text) -> (Rel)
  (+ Val X = Kampe*(Tips ? (#.Uge=uge))
     in
     if |X ? (#.HjHold=hold)| > 0 -> Rel(Tup(Bane:"hjemme"))
      & true -> Rel(Tup(Bane:"ude"))
     fi
  +)
end
```

Vi har i ovenstående eksempel benyttet os af operatoren `|R|`, som angiver størrelsen af relationen `R`, det vil sige, antallet af tupler i `R`.

Med introduktionen af tupeloperatorer og forall-operatoren kan vi nu skrive følgende alternative udgave af eksempel 14. Bemærk, at vi her bruger `Val`-mekanismen i indskudte udtryk til at introducere *korte* navne for attributudtrykkene `#.HjScore` og `#.UdeScore`.

Eksempel 22

```
Func (ugenr:Int) -> (Rel)
  (+ Val Strip =
       Func (uge:Int) -> (Rel)
         ((Turnering*Tips) ? (#.Uge=uge)) |- Id,Uge
       end
     Val X = Strip(ugenr)
     in !(X): (+ Val H = #.HjScore
                 Val U = #.UdeScore
                 in Rel(#\HjScore\UdeScore «
                           Tup(Tegn: if U<H -> "1"
                                     & U=H -> "X"
                                     & U>H -> "2"
                                     fi
                               )
                     )
              +)
  +)
end
```

Som det fremgår af det foregående, kan man både navngive og gemme relationer og funktioner, og man kan også give dem lokale navne ved hjælp af `Val`-mekanismen i indskudte udtryk, eller ved hjælp af formelle parametre i funktioner. Der er imidlertid ikke kun relationer og funktioner, der nyder denne bevågenhed – alle RASMUS typer kan benyttes på tilsvarende vis. Dette betyder, at udtryk af de forskellige typer kan konstrueres og komponeres på helt ensartet vis, hvilket også kan udtrykkes som, at "alle typer er førsteklasses borgere".

Som nævnt har RASMUS typerne `Bool`, `Int`, `Text`, `Tup`, `Rel` og `Func`. Ethvert udtryk har en type, nemlig typen af den værdi, der bliver resultatet.

5.5 Tekstudtryk

Vi har tidligere set eksempler på udtryk af samtlige typer – dog har vi for `Text`-udtrykkene kun set konstanter som `"AGF"`. Der findes imidlertid også `Text`-operatorer, som illustreret ved følgende eksempel på en funktion, der sætter årstal på de datoangivelser, der findes i `Spilledag` relationen.

Eksempel 23

```
Func (år:Text, dato:Text) -> (Text)
  if |dato|=4 -> år++dato
   & |dato|=3 -> år++"0"++dato
  fi
end
```

Hvis vi kalder denne funktion for `LangDato` er værdien af anvendelsen

```
Langdato("96","325")
```

lig med `960325`. Operatoren `++` sammensætter *konkatenerer* altså to tegnfølger, og `|t|` returnerer længden af en tegnfølge `t`. Man kan også udtage delfølger af tegnfølger. Værdien af udtrykket

```
"325"(0..2)
```

er således `"32"`, idet udtrykket `t(i..j)` i almindelighed betegner den delfølge af `t`, der starter i det `i`'te tegn (nummereret fra 0) og ender i det `(j-1)`'te tegn. Følgende eksempel illustrerer nu brugen af hjælpefunktionen `LangDato` i et forall-udtryk.

Eksempel 24

Skab en relation magen til `Spilledag`, men hvor datoangivelserne nu er komplette

```
!(Spilledag): Rel(# « Tup(Dato: LangDato("91",#.Dato)))
```

Værdien af udtrykket er

Kamp:Int	Dato:Text
3	910401
19	910520
20	910520
10	910421
23	910530
24	910602
25	910602
13	910512
14	910512
15	910512
7	910414
8	910414
4	910324
21	910523
11	910505
12	910505
1	910316
29	910623
30	910623
5	910407
6	910407
2	910317
16	910516
17	910516
18	910516
22	910526
9	910428
26	910609
27	910609
28	910619

5.6 Skema- og typecheck

Når man skriver funktioner, der overfører relationer som parametre, så kender man selvsagt ikke skemaerne for de relationer, der optræder som *aktuelle* parametre, når funktionen anvendes. RASMUS indeholder derfor en facilitet, der tillader at spørge om en relation har et givet attributnavn. Følgende eksempel illustrerer dette.

Eksempel 25

Skriv en funktion, der for en given `Kamp`-relation og et givet hold returnerer de kampe, hvor det pågældende hold er på hjemmebane.

```
Func (K:Rel, hold:Text) -> (Rel)
  if has(K,HjHold) -> K ? (#.HjHold=hold)
   & true -> zero
  fi
end
```

Udtrykket `has(R,A)` giver `true`, hvis relationen `R` har attributten `A` og ellers `false`. Som eksemplet viser, kan man bruge dette til at undgå at løbe ind i fejl under eventuel anvendelse af en funktion på et argument, der ikke har det rigtige skema. I eksemplet returneres den tomme relation, angivet af konstanten `zero`, hvis argumentet ikke "passer", men man kunne naturligvis også returnerer en relation, der indeholder en passende advarsel.

Man kan også undersøge, om en relation har den rigtige type. Hvis denne facilitet anvendes i eksempel 25, får funktionen følgende udseende.

```
Func (K:Rel, hold:Text) -> (Rel)
  if has(K,HjHold) and is-Text(K,HjHold) -> K ? (#.HjHold=hold)
   & true -> zero
  fi
end
```

Udtrykket `is-Text(R,A)` undersøger om attributten `A` i relationen `R` har type Text. Der findes tilsvarende funktioner `is-Int` og `is-Bool`.

5.7 Aggregeringsfunktioner

Som det sidste eksempel i denne "fodbold-Odyssé" viser vi nu, hvordan man beregner en relation med skema

Hold:Text	Points:Int

der for hver af de seks deltagende klubber angiver, hvor mange point klubben fik i turneringen. Til brug herfor skal vi introducere *aggregeringsfunktioner*, som tillader at foretage diverse beregninger på samtlige værdier af en given attribut i en relation. Følgende simple eksempel illustrerer dette.

Eksempel 26

Angiv, hvor mange mål der blev scoret på udebane i turneringen

```
add(Resultater,UdeScore)
```

Værdien af udtrykket er `33`. ∎

Aggregeringsfunktionen `add(R,A)` adderer værdierne af attributten `A` i relationen `R`. De øvrige aggregeringsfunktioner er

```
count(R,A), mult(R,A), max(R,A), min(R,A)
```

og de henholdsvis tæller, multiplicerer, maksimerer og minimerer `A`-værdierne i `R`.

Eksempel 27

Skab en relation, som for hver kamp angiver hvor mange point de to hold fik i den pågældende kamp

```
!(Turnering):
    (+ Val H = #.HjScore
       Val U = #.UdeScore
       in if H<U -> Rel(Tup(Id:#.Id, Hold:#.UdeHold, Points:2))
           & H=U -> Rel(Tup(Id:#.Id, Hold:#.UdeHold, Points:1)) +
                    Rel(Tup(Id:#.Id, Hold:#.HjHold, Points:1))
           & H>U -> Rel(Tup(Id:#.Id, Hold:#.HjHold, Points:2))
          fi
    +)
```

Værdien er

Id:Int	Hold:Text	Points:Int
6	Frem	1
6	Lyngby	1
25	Frem	2
10	AGF	2
23	AGF	1
23	Frem	1
12	AaB	1
12	OB	1
30	AaB	2
11	Lyngby	2
29	Brøndby	1
29	Lyngby	1
1	Brøndby	2
19	AGF	1
19	Brøndby	1
2	Frem	2
20	Frem	1
20	OB	1
13	AaB	2
16	Frem	2
15	Brøndby	1
15	OB	1
18	OB	1
18	Brøndby	1
5	AaB	1
5	Brøndby	1
24	Brøndby	2
8	Brøndby	1
8	Frem	1
27	Brøndby	2
14	Lyngby	2
17	Lyngby	1
17	AGF	1
9	Lyngby	2
28	Lyngby	1
28	OB	1
4	AGF	1
4	OB	1
21	AGF	2
3	Lyngby	2
22	Lyngby	2
7	AaB	2
26	AGF	2

Hvis vi gemmer denne relation under navnet `EnkeltPoints` får vi det ønske-

de resultat på følgende måde.

Eksempel 28

```
(+ Val AGF = EnkeltPoints ? (#.Hold="AGF")
   Val OB = EnkeltPoints ? (#.Hold="OB")
   Val AaB = EnkeltPoints ? (#.Hold="AaB")
   Val Lyngby = EnkeltPoints ? (#.Hold="Lyngby")
   Val Brøndby = EnkeltPoints ? (#.Hold="Brøndby")
   Val Frem = EnkeltPoints ? (#.Hold="Frem")
   in Rel(Tup(Hold:"AGF", Points:add(AGF,Points)) +
      Rel(Tup(Hold:"OB", Points:add(OB,Points)) +
      Rel(Tup(Hold:"AaB", Points:add(AaB,Points)) +
      Rel(Tup(Hold:"Lyngby", Points:add(Lyngby,Points)) +
      Rel(Tup(Hold:"Brøndby", Points:add(Brøndby,Points)) +
      Rel(Tup(Hold:"Frem", Points:add(Frem,Points))
+)
```

Værdien er

Hold:Text	Points:Int
Lyngby	14
AGF	10
OB	6
AaB	8
Frem	10
Brøndby	12

5.8 Factor

Denne måde at udregne turneringsresultatet er ikke netop elegant, og hvis `Hold`-attributten indeholdt flere end seks forskellige værdier, eller – endnu værre – hvis vi ikke kendte værdierne på forhånd, så er fremgangsmåden slet ikke brugbar. Derfor indeholder RASMUS en mere generel facilitet, den såkaldte *factor*-operator, som tillader at udføre denne type beregninger på mere hensigtsmæssig vis. I dette tilfælde ville beregningen se ud som følger.

Eksempel 29

```
!(EnkeltPoints)|Hold: Rel(# « Tup(Points:add(@(1),Points)))
```

Lad os se, hvordan ovenstående fungerer. Argumentet til factor-beregningen er relationen `EnkeltPoints`. Desuden angiver man et attributnavn, i dette tilfælde `Hold`. På baggrund heraf beregnes en række *faktorpar* af formen $(t_1, r_1), (t_2, r_2), \ldots, (t_k, r_k)$. Tuplerne $t_1, t_2, \ldots, t_k$ er netop de, der findes i relationen `EnkeltPoints|+Hold`. Relationen r_i er netop `(EnkeltPoints*Rel(`t_i`))|-Hold`. Det vil sige, der gælder følgende ligning:

$$\texttt{Enkeltpoints} = \sum_{i=1}^{k} \texttt{Rel(}t_i\texttt{)*}r_i$$

For hvert af disse faktorpar beregnes højresiden af factor-udtrykket med `#` erstattet med t_i og `@(1)` erstattet med r_i. Det endelige resultat er foreningen af disse delresultater. For ovenstående eksempel har vi følgende seks faktorpar.

Lyngby

Id:Int	Points:Int
6	1
11	2
29	1
14	2
17	1
9	2
28	1
3	2
22	2

AGF

Id:Int	Points:Int
10	2
23	1
19	1
17	1
4	1
21	2
26	2

OB

Id:Int	Points:Int
12	1
20	1
15	1
18	1
28	1
4	1

AaB

Id:Int	Points:Int
12	1
30	2
13	2
5	1
7	2

Frem

Id:Int	Points:Int
6	1
25	2
23	1
2	2
20	1
16	2
8	1

Brøndby

Id:Int	Points:Int
29	1
1	2
19	1
15	1
18	1
5	1
24	2
8	1
27	2

Det skulle nu være klart, hvorledes vi beregner det ønskede resultat.

Et factor-udtryk kan involvere flere attributnavne. I så fald indeholder faktor-parrets tupel alle disse attributter og relationerne er tilsvarende smallere.

Eksempel 30

Skab en relation, der for hver mulig score fortæller, hvor mange kampe der fik denne score.

```
!(Turnering)|HjScore,UdeScore: Rel(# « Tup(Antal: |@(1)|))
```

Værdien af udtrykket er

HjScore:Int	UdeScore:Int	Antal:Int
0	0	3
1	0	1
1	1	8
2	0	1
0	2	1
1	2	4
3	0	1
2	1	3
0	3	1
2	2	2
3	1	1
4	1	2
5	1	1
6	1	1

■

Det skulle nu let kunne ses, at forall-operatoren blot er et specialtilfælde af factor, hvor man angiver *samtlige* attributnavne. I så tilfælde består faktorparrene blot af alle relationens tupler parret med relationen `one`, der indeholder et enkelt tupel med tomt skema (som ikke bidrager med nogen interessant information).

Factor-operationen er dog endnu mere generel, idet man også kan angive flere relationer. I så fald skal de angivne attributnavne tilhøre skemaerne for samtlige de indgående relationer. Nu får man ikke blot nogle faktor*par*, men sekvenser bestående af et tupel og en delrelation for hver indgående relation. Delrelationen for den i'te relation benævnes `@(i)`. Hvis vi betragter et generelt factor-udtryk:

$$\texttt{!(R}_1\texttt{,...,R}_m\texttt{)|n}_1\texttt{,...,n}_l\texttt{: ...}$$

hvor den i'te af k faktorsekvenser er $(t_i, r_{i1}, \ldots, r_{im})$, så gælder følgende matrixligning:

$$(\texttt{R}_1, \ldots, \texttt{R}_m) = (\texttt{Rel}(t_1), \ldots, \texttt{Rel}(t_k)) \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1m} \\ r_{21} & r_{22} & \cdots & r_{2m} \\ \vdots & \vdots & & \vdots \\ r_{k1} & r_{k2} & \cdots & r_{km} \end{pmatrix}$$

Med denne udvidelse er factor nu så generel, at man kan nøjes med den som eneste operation (sammen med krydsprodukt af relationer med disjunkte skemaer).

5.9 Oversigt

Som tidligere nævnt er alle typer i RASMUS "første klasses borgere". Det betyder, at værdier af alle typer kan både navngives og gemmes, og at udtryk af alle typer kan være parametre til og resultater af funktioner. I det følgende giver vi en mere systematisk fremstilling af udseendet af de forskellige udtrykstyper. Vi betegner udtryk af type `Int`, `Bool`, `Text`, `Tup`, `Rel` og `Func` med henholdsvis $\mathcal{I}$, $\mathcal{B}$, $\mathcal{X}$, $\mathcal{T}$, $\mathcal{R}$ og $\mathcal{F}$. Et udtryk af en atomar type betegnes $\mathcal{A}$. Et udtryk af en *ordnet* type betegnes $\mathcal{O}$; alle typer pånær `Func` er ordnede og kan dermed sammenlignes med `=`, `<>`, `<`, `>`, `<=` og `>=`. Et udtryk af en vilkårlig type angives med $\mathcal{U}$. Et `Val-`, parameter- eller attributnavn betegnes med `n`.

Udtryk af type `Bool` ($\mathcal{B}$):

Betegnelse	Syntaks
konstant	`true`, `false`
standardværdi	`?-Bool`
navn	`n`
attribut	$\mathcal{T}$`.n`
sammenligning	$\mathcal{O}_1$`=`$\mathcal{O}_2$, $\mathcal{O}_1$`<>`$\mathcal{O}_2$, $\mathcal{O}_1$`<`$\mathcal{O}_2$, $\mathcal{O}_1$`>`$\mathcal{O}_2$, $\mathcal{O}_1$`<=`$\mathcal{O}_2$, $\mathcal{O}_1$`>=`$\mathcal{O}_2$
match	$\mathcal{T}_1$`~`$\mathcal{T}_2$
negation	`not` $\mathcal{B}$
konjunktion	$\mathcal{B}_1$ `and` $\mathcal{B}_2$
disjunktion	$\mathcal{B}_1$ `or` $\mathcal{B}_2$
skemacheck	`has(`$\mathcal{R}$`,n)`
typecheck	`is-Int(`$\mathcal{R}$`,n)`, `is-Bool(`$\mathcal{R}$`,n)`, `is-Text(`$\mathcal{R}$`,n)`
maksimum	`max(`$\mathcal{R}$`,n)`
minimum	`min(`$\mathcal{R}$`,n)`
funktionsanvendelse	$\mathcal{F}$`(`$\mathcal{U}_1$`,...,`$\mathcal{U}_k$`)`
indskudt udtryk	`(+ Val n`$_1$`=`$\mathcal{U}_1$ `...Val n`$_k$`=`$\mathcal{U}_k$ `in` $\mathcal{B}$ `+)`
betinget udtryk	`if` $\mathcal{B}$ `->` $\mathcal{B}$ `...fi`

Standardværdien `?-Bool` er en ekstra konstant af typen `Bool`, der blandt andet optræder som "ubestemte" værdier af `Bool`-attributter. Der findes også standardværdier af typerne `Int` og `Text`. Standardværdier omtales nærmere i næste kapitel.

Udtryk af type `Int` ($\mathcal{I}$):

Betegnelse	Syntaks
konstant	..., `-2`, `-1`, `0`, `1`, `2`, `3`, ...
standardværdi	`?-Int`
navn	`n`
attribut	$\mathcal{T}$`.n`
sum	$\mathcal{I}_1$ `+` $\mathcal{I}_2$
differens	$\mathcal{I}_1$ `-` $\mathcal{I}_2$
produkt	$\mathcal{I}_1$ `*` $\mathcal{I}_2$
kvotient	$\mathcal{I}_1$ `/` $\mathcal{I}_2$
rest	$\mathcal{I}_1$ `mod` $\mathcal{I}_2$
relationsstørrelse	`|`$\mathcal{R}$`|`
tekstlængde	`|`$\mathcal{X}$`|`
maksimum	`max(`$\mathcal{R}$`,n)`
minimum	`min(`$\mathcal{R}$`,n)`
addition	`add(`$\mathcal{R}$`,n)`
multiplikation	`mult(`$\mathcal{R}$`,n)`
optælling	`count(`$\mathcal{R}$`,n)`
funktionsanvendelse	$\mathcal{F}$`(`$\mathcal{U}_1$`,...,`$\mathcal{U}_k$`)`
indskudt udtryk	`(+ Val` n_1`=`$\mathcal{U}_1$ ... `Val` n_k`=`$\mathcal{U}_k$ `in` $\mathcal{I}$ `+)`
betinget udtryk	`if` $\mathcal{B}$ `->` $\mathcal{I}$ ... `fi`
antal dage	`days(`$\mathcal{X}_1$`,`$\mathcal{X}_2$`)`

Udtryk af type `Text` ($\mathcal{X}$):

Betegnelse	Syntaks
konstant	`""`, `"AGF"`, ...
standardværdi	`?-Text`
navn	`n`
attribut	$\mathcal{T}$`.n`
deltekst	$\mathcal{X}$`(`$\mathcal{I}_1$`..`$\mathcal{I}_2$`)`
konkatenation	$\mathcal{X}_1$ `++` $\mathcal{X}_2$
maksimum	`max(`$\mathcal{R}$`,n)`
minimum	`min(`$\mathcal{R}$`,n)`
funktionsanvendelse	$\mathcal{F}$`(`$\mathcal{U}_1$`,...,`$\mathcal{U}_k$`)`
indskudt udtryk	`(+ Val n`$_1$`=`$\mathcal{U}_1$ ... `Val n`$_k$`=`$\mathcal{U}_k$ `in` $\mathcal{X}$ `+)`
betinget udtryk	`if` $\mathcal{B}$ `->` $\mathcal{X}$ ... `fi`
præfiks	`before(`$\mathcal{X}_1$`,`$\mathcal{X}_2$`)`
suffiks	`after(`$\mathcal{X}_1$`,`$\mathcal{X}_2$`)`
dags dato	`today`
fremtidig dato	`date(`$\mathcal{X}$`,`$\mathcal{I}$`)`

Udtryk af type `Tup` ($\mathcal{T}$):

Betegnelse	Syntaks
konstant	`Tup(`$\mathtt{n}_1$`:`$\mathcal{A}_1$`,...,`$\mathtt{n}_k$`:`$\mathcal{A}_k$`)`
navn	`n`, `#`
opdatering	$\mathcal{T}_1$ `«` $\mathcal{T}_2$
elimination	$\mathcal{T}$ `\` `n`
funktionsanvendelse	$\mathcal{F}$`(`$\mathcal{U}_1$`,...,`$\mathcal{U}_k$`)`
indskudt udtryk	`(+ Val` $\mathtt{n}_1$`=`$\mathcal{U}_1$ `...Val` $\mathtt{n}_k$`=`$\mathcal{U}_k$ `in` $\mathcal{T}$ `+)`
betinget udtryk	`if` $\mathcal{B}$ `->` $\mathcal{T}$ `...fi`

Udtryk af type `Rel` ($\mathcal{R}$):

Betegnelse	Syntaks
konstant	`Rel(`$\mathcal{T}$`)`, `zero`
navn	`n`
union	$\mathcal{R}_1$ `+` $\mathcal{R}_2$
join	$\mathcal{R}_1$ `*` $\mathcal{R}_2$
difference	$\mathcal{R}_1$ `-` $\mathcal{R}_2$
project	$\mathcal{R}$ `|+` $\mathtt{n}_1$`,...,`$\mathtt{n}_k$, $\mathcal{R}$ `|-` $\mathtt{n}_1$`,...,`$\mathtt{n}_k$
select	$\mathcal{R}$ `?` $\mathcal{B}$
rename	$\mathcal{R}$ `[`$\mathtt{n}_1$ `<-` $\mathtt{n}_2$`]`
forall	`!(`$\mathcal{R}_1$`):` $\mathcal{R}_2$
factor	`!(`$\mathcal{R}_1,\ldots,\mathcal{R}_m$`)|`$\mathtt{n}_1$`,...,`$\mathtt{n}_k$`:` $\mathcal{R}$
funktionsanvendelse	$\mathcal{F}$`(`$\mathcal{U}_1$`,...,`$\mathcal{U}_k$`)`
indskudt udtryk	`(+ Val` $\mathtt{n}_1$`=`$\mathcal{U}_1$ `...Val` $\mathtt{n}_k$`=`$\mathcal{U}_k$ `in` $\mathcal{R}$ `+)`
betinget udtryk	`if` $\mathcal{B}$ `->` $\mathcal{R}$ `...fi`

Udtryk af type `Func` ($\mathcal{F}$):

Betegnelse	Syntaks
konstant	`Func (...) -> (...)` $\mathcal{U}$ `end`
navn	`n`
funktionsanvendelse	$\mathcal{F}$`(`$\mathcal{U}_1$`,...,`$\mathcal{U}_k$`)`
indskudt udtryk	`(+ Val` $\mathtt{n}_1$`=`$\mathcal{U}_1$ `...Val` $\mathtt{n}_k$`=`$\mathcal{U}_k$ `in` $\mathcal{F}$ `+)`
betinget udtryk	`if` $\mathcal{B}$ `->` $\mathcal{F}$ `...fi`

6 Overordnede begreber

Efter de foregående kapitlers eksempelbaserede præsentation skal vi nu give en mere systematisk og begrebsorienteret beskrivelse af RASMUS.

6.1 Relationelle databaser

RASMUS er interessant af mindst to årsager, dels fordi sproget indeholder de væsentligste ingredienser fra det emneområde, der hedder *relationelle databaser*, og dels fordi det repræsenterer nogle af de mest fundamentale begreber, der findes i programmeringssprog og programmeringsnotationer.

Relationelle databaser blev "opfundet" omkring 1970, som et forslag til en "ren", systematisk og teoretisk velfunderet teknik til opbevaring og behandling af store datamængder. Den basale iagttagelse var, at dels kan praktisk talt enhver samling informationer, man kan forestille sig, repræsenteres i form af en eller flere relationer, og dels tillader de basale relationelle operatorer (union, join, select, project), at man kombinerer og udtrækker informationer fra vilkårlige relationer på en meget generel måde.

Dette betyder blandt andet, at hvis information opbevares i form af relationer, så kan forskellige data, der fra fødslen ikke synes at have meget med hinanden at gøre, bearbejdes af det samme *databehandlingssystem*, uanset tidspunktet for og omstændighederne ved det tilblivelse. Dette giver en fleksibilitet og en generalitet, der står i skarp kontrast til en strategi, hvor man designer både datarepræsentation og tilhørende programmer afhængigt af den konkrete opgave og uden at tage højde for, at det senere skal kunne "samkøres" med andre data. Relationelle databaser introducerer derfor ikke blot nogle nyttige begreber og en hensigtsmæssig systematik, teknikken sparer også ressourcer i forbindelse med praktisk databehandling.

6.2 Typer, værdier og udtryk

Uanset den praktiske nytteværdi, som bestemt er til stede, skal vi i det følgende koncentere os om det andet interessante aspekt, nemlig RASMUS som programmeringssprog.

For at forstå et programmeringssprog, herunder RASMUS, er det afgørende at forstå det centrale begreb *værdi*, som er en temmelig abstrakt størrelse. Stærkt

forenklet kan man sige, at faciliteterne i RASMUS er beregnet til at *skabe*, *gemme* og *genfinde* værdier: heltal, sandhedsværdier, tegnfølger, tupler, relationer og funktioner. Værdierne skabes som resultater af udtryk, hvorfor udtryk også klassificeres efter hvilken slags værdier, de producerer. I datalogisk terminologi bruger man *typer*, det vil sige, at vi i RASMUS har værdier og udtryk af typerne: `Int`, `Bool`, `Text`, `Tup`, `Rel` og `Func`.

6.3 Konstante udtryk

Ethvert udtryk skaber (eller producerer) altså en værdi af den pågældende type. De simpleste udtryk er *konstanterne*, som fx

```
5
```

Dette er et udtryk af type `Int`, der som resultat (eller værdi) har *tallet* 5. Bemærk, at *tallet* 5 i sig selv er noget abstrakt og at udtrykket `5` blot er et af mange mulige udtryk, hvis værdi er dette tal. Hvis man fx var forelsket i romertal eller binære tal, så kunne man bruge `V` eller `101` som udtryk med værdi 5. I RASMUS benyttes den velkendte decimale notation for konstanter af type `Int`.

De øvrige typer har også konstanter, det vil sige udtryk, der angiver deres værdier. Som nævnt i kapitel 4 er følgende eksempler karakteristiske for konstante udtryk af de forskellige typer

```
Text:

Bool: true

Text: "AGF"

Tup: Tup(hold:"AGF")

Rel: Rel(Tup(hold:"AGF"))

Func: Func () -> (Text) "AGF" end
```

Deres værdier er henholdsvis

- sandhedsværdien sand
- tegnfølgen bestående af tegnene `A`, `G` og `F`

- et tupel med attribut `hold` hvis tilsvarende værdi er tegnfølgen bestående af tegnene `A`, `G` og `F`
- en relation indeholdende et enkelt tupel med attribut `hold` og tilsvarende værdi tegnfølgen bestående af tegnene `A`, `G` og `F`
- en funktion uden parametre, der returnerer tegnfølgen bestående af tegnene `A`, `G` og `F`

Enhver type har en *mængde* af værdier knyttet til sig. Værdimængderne i RASMUS er

`Text:`

`Int:` mængden af heltal

`Bool:` de to sandhedsværdier

`Text:` mængden af følger af ASCII tegn

`Tup:` mængden af tupler

`Rel:` mængden af relationer, der hver er en endelig mængde af tupler med samme skema

`Func:` mængden af funktioner med nul eller flere argumenter af en af de seks typer og med et resultat, der er af en af de seks typer

Værdimængden i `Int`, `Bool`, `Text`, `Tup` og `Rel` er *ordnede*, det vil sige, man kan sammenligne deres værdier. Ordningerne er som følger

`Text:`

`Int:` den sædvanlige ordning på heltallene

`Bool:` `false` er mindre end `true`

`Text:` alfabetisk ordning, hvor "bogstaverne" er ASCII tegn i den sædvanlige rækkefølge

`Tup:` $\texttt{t}_1$ er mindre end $\texttt{t}_2$, hvis alle attributnavne i $\texttt{t}_1$ også forekommer i $\texttt{t}_2$ og værdier hørende til fælles attributnavne er ens i de to tupler.

`Rel:` $\texttt{r}_1$ er mindre end $\texttt{r}_2$, hvis de har samme skema, og alle tupler i $\texttt{r}_1$ også forekommer i $\texttt{r}_2$

For de tre atomare typer (`Int`, `Bool` og `Text`) er dette imidlertid ikke hele sandheden. De tre typer, hvis værdier er de eneste mulige attributværdier, har endnu en værdi, nemlig den såkaldte *standardværdi*. Disse standardværdier kan angives med konstantudtrykkene `?-Int`, `?-Bool` og `?-Text`. Meningen med og behovet for sådanne standardværdier kan illustreres af følgende udvidelse af informationen om vore fodboldkampe. Relationen `Tips` indeholder kun kampe fra `Runde1`, den første halvdel af turneringen. Dette skyldes, at spilledagene i anden halvdel falder på en sådan måde, at de seks tophold ikke spiller inden for samme uge på en sådan måde, at vi kan definere en tipskupon med tre kampe, hvor alle hold deltager. Hvis man nu tillader andre kampe at optræde på kuponen, er det naturligvis nemt også at lave tipskuponer for `Runde2`. I så falde kunne `Tips` relationen have følgende udseende

Id:Int	Uge:Int	Nr:Int
⋮	⋮	⋮
12	18	3
13	19	1
14	19	2
15	19	3
19	21	1
20	21	2
▒▒▒▒▒	21	3
▒▒▒▒▒	23	1
26	23	2
27	23	3
29	25	1
30	25	1
▒▒▒▒▒	25	3

Som det ses, er der nogle af kampene for hvilke `Id`-feltet er "tomt". Grunden er, at vi kun har Superliga kampe i vores database, det vil sige, vi mangler identifikationen af den relevante kamp. Nu kan felter i tupler imidlertid ikke være tomme – RASMUS forlanger, at der altid er knyttet en værdi til en attribut – og derfor skal vi bruge en værdi, der angiver "manglende information". Dette er præcis hvad standardværdien kan bruges til. De "tomme felter" er altså i virkeligheden ikke tomme; de indeholder `?-Int`, standardværdien for typen `Int`. I RASMUS systemet angives dette ved, at feltet er *lysegråt*. Vi kan illustrere sammenhængen mellem standardværdien `?-Int` og et lysegråt felt ved hjælp af følgende eksempel.

Eksempel 31

Værdien af udtrykket

```
Rel(Tup(Hovsa:?-Int))
```

er

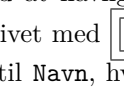
Udover at angive manglende information kan standardværdien også bruges til at angive at et felt er irrelevant for et givent tupel, eller at der er tale om andre former for undtagelser. Vi bemærker, at et tom *hvidt* felt ikke angiver en standardværdi men derimod en `Text`-attribut, hvis værdi er den tomme tegnfølge.

6.4 Navne og tilstande

Vi vender nu tilbage til beskrivelsen af udtryk, hvor vi allerede har set, at konstanter er udtryk. En anden form for simple udtryk er navne, som fx

```
Kampe
```

Værdien af et sådant udtryk er den værdi, der er *bundet* til det pågældende navn. I eksemplet er det den relation, der indeholder samtlige kampe i turneringen. Når man introducerer et navn i RASMUS er det altid for at navngive en værdi af en af de seks typer, det vil sige, alle navne i RASMUS har altid en værdi bundet til sig. Den samling af navne, med tilhørende værdier, man på et givet tidspunkt har adgang til, kaldes under et for den aktuelle *tilstand*, og udregningen af udtryk i RASMUS foregår altid relativt til en sådan tilstand.

Der findes tre måder, hvorpå man kan binde værdier til navne og derved modificere den aktuelle tilstand. Den første måde er ved at navngive et udtryk (ved hjælp af **Save last**) hvilket i eksemplerne er angivet med 

. Effekten heraf er, at den pågældende værdi bliver bundet til `Navn`, hvorefter denne binding tilføjes til tilstanden. Hvis der allerede findes en binding til `Navn`, så forsvinder denne. Vi kalder en sådan udvidelse/overskrivning for en *opdatering* af tilstanden.

Den anden måde at introducere bindinger er ved hjælp af et indskudt udtryk af formen

```
(+  Val id1 = udtryk1
    Val id2 = udtryk2
    ⋮
    Val idn = udtrykn
    in udtryk
+)
```

Her sker der følgende. Først udregnes **udtryk$_1$** i den aktuelle tilstand og dets værdi bindes til **id$_1$**, hvorefter den aktuelle tilstand opdateres med denne binding. Derefter udregnes **udtryk$_2$** i den nye tilstand, værdien bindes til **id$_2$** og tilstanden opdateres med denne binding. Dette forsætter indtil alle **n** opdateringer er foretaget, hvorefter **udtryk** udregnes i den resulterende tilstand. Herefter *retableres* den oprindelige tilstand, det vil sige, udregningen af et indskudt udtryk påvirker således ikke tilstanden – de udførte opdateringer er *lokale* og *midlertidige*.

Den sidste måde at foretage bindinger på er i forbindelse med funktionsanvendelse. Hvis funktionen

```
Func (p1:T1, p2:T2) -> (T)
  udtryk
end
```

er bundet til navnet `F` i den aktuelle tilstand, vil funktionsanvendelsen

`F(`**udtryk$_1$**`,`**udtryk$_2$**`)`

bevirke følgende. Først udregnes værdierne af **udtryk$_1$** og **udtryk$_2$** i den aktuelle tilstand. Dernæst bindes disse værdier til henholdsvis `p1` og `p2`, og tilstanden opdateres med disse to bindinger. Nu udregnes **udtryk** i den således modificerede tilstand, og den resulterende værdi er også værdien af funktionsanvendelsen. Endelig retableres den oprindelige tilstand. Udregningen af en funktionsanvendelse påvirker altså heller ikke tilstanden.

6.5 Sammensatte udtryk

Efter at have introduceret simple udtryk (konstanter og navne) og tilstande, skal vi nu se på *sammensatte* udtryk, der fås ved på sædvanlig algebraisk vis at sammensætte to deludtryk ved hjælp af en *operator*. Et simpelt eksempel er

```
3+4
```

hvor operatoren `+` er heltalsaddition og de to deludtryk er konstanter af type `Int`. Det er klart, at `+` stiller det krav, at operanderne skal være heltal. Fx er udtrykket

```
3+"AGF"
```

hverken tilladt eller særligt meningsfuldt, idet det naturligvis ikke giver mening at addere tallet `3` og tegnfølgen `A`, `G`, `F`. Om et udtryk som

```
3+x
```

er tilladt, afhænger af typen af den værdi, der er bundet til `x`. Hvis den er af type `Int` er sagen i orden, ellers er det en fejl.

Vi gennemgår nu udtrykkene i oversigten i afsnit 5.8 og kommenterer i nødvendigt omfang operatorernes betydning, samt hvilke krav de stiller til deres operander.

For samtlige seks udtrykstyper gælder følgende.

Konstanter:
Taler for sig selv. Resultatet er den "oplagte" (abstrakte) værdi.

Navne:
Et navn skal findes i tilstanden, og værdien er den, der her er bundet til navnet.

Funktionsanvendelse $\mathcal{F}$`(`$\mathcal{U}_1$`,...,`$\mathcal{U}_k$`)`*:*
$\mathcal{U}_1, \ldots, \mathcal{U}_k$ skal være udtryk af samme typer som de tilsvarende formelle parametre i den funktion, der er værdien af $\mathcal{F}$. Resultatet er som beskrevet i afsnit 6.4.

Indskudt udtryk `(+ Val ...in` $\mathcal{U}$ `+)`*:*
Som beskrevet i afsnit 6.4.

Betinget udtryk `if` $\mathcal{B}_1$ `->` $\mathcal{U}_1$ `&` $\ldots$ `&` $\mathcal{B}_n$ `->` $\mathcal{U}_n$ `fi`*:*
Resultatet opnås ved først at udregne $\mathcal{B}_1, \ldots, \mathcal{B}_n$ i rækkefølge indtil det første $\mathcal{B}_i$, hvis værdi er sand. Resultatet er da værdien af $\mathcal{U}_i$. Hvis intet $\mathcal{B}_i$ er sandt, så er resultatet (lidt arbitrært) tallet 0.

For de *atomare typer* **gælder følgende.**

Standardværdien `?-`$\mathcal{A}$*:*
Dette er en "ekstra" værdi, der tilføjes til typen. Den er kun lig med sig selv og står ikke i nogen ordningsrelation til de andre værdier. Standardværdien må kun være operand til sammenligningsoperatorer.

Attribut $\mathcal{T}$`.n`*:*

Navnet `n` skal være en attribut i værdien af $\mathcal{T}$. Resultatet er den tilsvarende attributværdi.

Maksimum `max(`$\mathcal{R}$`,n)`*:*
Navnet `n` skal være en attribut i værdien af $\mathcal{R}$. Resultatet er den største `n`-værdi i $\mathcal{R}$ i den pågældende types ordning. Hvis $\mathcal{R}$ er tom, eller `n`-attributten overalt har standardværdien `?-`$\mathcal{A}$, er resultatet denne standardværdi.

Minimum `min(`$\mathcal{R}$`,n)`*:*
Analog til maksimum.

For `Bool`-udtryk gælder følgende.

Sammenligning $\mathcal{O}_1$`=`$\mathcal{O}_2$*,* $\mathcal{O}_1$`<`$\mathcal{O}_2$*, og så videre:*
$\mathcal{O}_1$ og $\mathcal{O}_2$ skal være af samme type. Resultatet er `true`, hvis værdierne har den angivne indbyrdes ordning, og `false` ellers.

Negation `not` $\mathcal{B}$*:*
Resultatet er `true` (`false`) hvis værdien af $\mathcal{B}$ er `false` (`true`).

Konjunktion $\mathcal{B}_1$ `and` $\mathcal{B}_2$*:*
Resultatet er `true`, hvis både $\mathcal{B}_1$ og $\mathcal{B}_2$ har værdi `true`, og `false` ellers.

Disjunktion $\mathcal{B}_1$ `or` $\mathcal{B}_2$*:*
Resultatet er `false`, hvis både $\mathcal{B}_1$ og $\mathcal{B}_2$ har værdi `false`, og `true` ellers.

Skemacheck `has(`$\mathcal{R}$`,n)`*:*
Resultatet er `true`, hvis `n` er attributnavn i $\mathcal{R}$, og `false` ellers.

Typecheck `is-`$\mathcal{A}$`(`$\mathcal{R}$`,n)`*:*
Resultatet er `true`, hvis `n`'s type i $\mathcal{R}$'s type er $\mathcal{A}$, og `false` ellers.

Match $\mathcal{X}_1$`~`$\mathcal{X}_2$*:*
Resultatet er `true`, hvis $\mathcal{X}_1$ optræder som en del af $\mathcal{X}_2$.

For `Int`-udtryk gælder følgende.

Regneoperatorer `+`*,* `-`*,* `*`*,* `/`*,* `mod`*:*
Taler for sig selv.

Relationsstørrelse og tekstlængde, `|`$\mathcal{R}$`|` *og* `|`$\mathcal{X}$`|`*:*
Resultatet er antallet af tupler i $\mathcal{R}$ og antallet af tegn i $\mathcal{X}$.

Addition `add(`$\mathcal{R}$`,n)`*:*

Navnet `n` skal være en attribut i $\mathcal{R}$ af type `Int`. Resultatet er summen af `n`-værdierne i $\mathcal{R}$. Hvis $\mathcal{R}$ er tom, eller `n`-værdierne alle er `?-Int`, er resultatet `?-Int`.

Multiplikation `mult(`$\mathcal{R}$`,n)`*:*
Analog til addition.

Optælling `count(`$\mathcal{R}$`,n)`*:*
Navnet `n` skal være en attribut i $\mathcal{R}$. Resultatet er antallet af `n`-værdierne i $\mathcal{R}$, der er forskellige fra standardværdien.

Antal dage `days(`$\mathcal{T}_1$`,`$\mathcal{T}_2$`)`*:*
Resultatet er antallet af dage mellem datoerne $\mathcal{T}_1$ og $\mathcal{T}_2$. En dato repræsenteres som en tekst på formen `dd/mm/åå`.

For `Text`-udtryk gælder følgende.

Deltekst $\mathcal{X}$`(`$\mathcal{I}_1$`..`$\mathcal{I}_2$`)`*:*
Resultatet er delteksten af $\mathcal{X}$ bestående af de tegn, hvis indices er større end eller lig med værdien af $\mathcal{I}_1$ om mindre end værdien af $\mathcal{I}_2$.

Konkatenation $\mathcal{X}_1$ `++` $\mathcal{X}_2$*:*
Resultatet er sammensætningen af værdierne af $\mathcal{X}_1$ og $\mathcal{X}_2$.

Præfiks `before(`$\mathcal{X}_1$`,`$\mathcal{X}_2$`)`*:*
Resultatet er den del af $\mathcal{X}_2$, der står foran første forekomst af delteksten $\mathcal{X}_1$. Hvis $\mathcal{X}_1$ slet ikke forekommer, så er resultatet den tomme tekst.

Suffiks `after(`$\mathcal{X}_1$`,`$\mathcal{X}_2$`)`*:*
Resultatet er den del af $\mathcal{X}_2$, der står efter første forekomst af delteksten $\mathcal{X}_1$. Hvis $\mathcal{X}_1$ slet ikke forekommer, så er resultatet den tomme tekst.

Dags dato `today`*:*
Resultatet er dags dato på formen `dd/mm/åå`.

Fremtidig dato `date(`$\mathcal{X}$`,`$\mathcal{I}$`)`*:*
Resultatet er den dato, der ligger $\mathcal{I}$ dage senere end datoen $\mathcal{X}$. En dato repræsenteres som en tekst på formen `dd/mm/åå`.

For `Tup`-udtryk gælder følgende.

Opdatering $\mathcal{T}_1 \ll \mathcal{T}_2$*:*
Resultatet er værdien af $\mathcal{T}_1$ opdateret med bindingerne i værdien af $\mathcal{T}_2$.

Elimination $\mathcal{T}$ `\ n`*:*
Resultatet er værdien af $\mathcal{T}$ med en eventuel binding til `n` fjernet.

For `Rel`-udtryk gælder følgende.

Konstanten `zero`*:*
Resultatet er den tomme relation med tomt skema.

Union og difference, $\mathcal{R}_1$ `+` $\mathcal{R}_2$ *og* $\mathcal{R}_1$ `-` $\mathcal{R}_2$*:*
$\mathcal{R}_1$ og $\mathcal{R}_2$ skal have samme skema. Resultatet er henholdsvis foreningsmængde og mængdedifferens af tuplerne i $\mathcal{R}_1$ og $\mathcal{R}_2$.

Project $\mathcal{R}$ `|+` $\mathtt{n}_1\mathtt{,}\ldots\mathtt{,}\mathtt{n}_k$ Navnene $\mathtt{n}_1, \ldots, \mathtt{n}_k$ skal være attributter i $\mathcal{R}$. Resultatet er mængden af tupler i $\mathcal{R}$ begrænset til disse attributter.

Select $\mathcal{R}$ `?` $\mathcal{B}$*:*
Resultatet er den delmængde af tupler i $\mathcal{R}$, hvor $\mathcal{B}$ har værdi `true`, når symbolet `#` erstattes med det pågældende tupel.

Rename $\mathcal{R}$ `[`$\mathtt{n}_1$ `<-` $\mathtt{n}_2$`]`*:*
Navnet $\mathtt{n}_1$ skal være en attribut i $\mathcal{R}$, hvorimod navnet $\mathtt{n}_2$ ikke må være det. Resultatet er mængden af tupler i $\mathcal{R}$ med $\mathtt{n}_1$ omdøbt til $\mathtt{n}_2$.

Join $\mathcal{R}_1$ `*` $\mathcal{R}_2$*:*
Fælles attributter i $\mathcal{R}_1$ og $\mathcal{R}_2$ skal have samme type. Resultatets skema er foreningen af de to skemaer. Resultatets tupler er alle de, der projiceret på $\mathcal{R}_1$'s skema tilhører $\mathcal{R}_1$, og projiceret på $\mathcal{R}_2$'s skema tilhører $\mathcal{R}_2$.

Forall `!(`$\mathcal{R}_1$`):` $\mathcal{R}_2$*:*
Resultatet er foreningsmængden af de relationer, der fremkommer ved for hvert $\mathcal{R}_1$-tupel at udregne $\mathcal{R}_2$ med symbolet `#` erstattet med tuplet.

Factor `!(`$\mathcal{R}_1, \ldots, \mathcal{R}_m$`)|`$\mathtt{n}_1\mathtt{,}\ldots\mathtt{,}\mathtt{n}_k$`:`$\mathcal{R}$*:*
Resultatet er foreningsmængden af de relationer, der fremkommer ved for hvert faktorsekvens $(t_i, r_{i1}, \ldots, r_{im})$ at beregne $\mathcal{R}$ med symbolet `#` erstattet med t_i og symbolet `@(j)` erstattet med r_{ij}.

For `Func`-udtryk gælder følgende.

Konstant `Func (...) -> (...)` $\mathcal{U}$ `end`*:*
Typen af $\mathcal{U}$ skal være resultattypen.

6.6 Algebraiske love

Som det fremgår af det foregående, spiller operatorerne den helt centrale rolle i RASMUS. Vi skal derfor i resten af dette kapitel interessere os lidt mere for deres *algebraiske* egenskaber. Hermed menes de love, der gælder for operatorerne, og som vi kan illusterer med følgende eksempel. Hvis `+` er almindelig addition af heltal, er der ingen der undrer sig over følgende udtryk

```
x + y + z
```

uanset, at det faktisk er *tvetydigt*, da det kan opfattes som `x + (y + z)` eller som `(x + y) + z`. Grunden til at vi ikke bliver forvirrede er, at det er ligegyldigt, hvilket af de to alternativer vi vælger, fordi `+` opfylder den *associative* lov. Denne lov udtrykker netop at de to alternativer er lige gode, det vil sige, for alle `x`, `y` og `z` gælder

```
x + (y + z) = (x + y) + z
```

Udover associativitet benytter vi os også af de *kommutative* og *distributive* love for heltalsaritmetik. Mere præcist gælder der for addition og multiplikation af heltal følgende love

Associativitet:	`x + (y + z) = (x + y) + z`
	`x * (y * z) = (x * y) * z`
Kommutativitet:	`x + y = y + x`
	`x * y = y * x`
Distributivitet:	`x * (y + z) = (x * y) + (x * z)`

Det interessante er nu, at hvis `+` og `*` i stedet betyder union og join af relationer, så gælder ovenstående fem love stadig. Ikke nok med det, der gælder yderligere følgende sammenhænge mellem union, join, select og project (hvor vi antager, at begge sider af ligningerne er lovlige udtryk)

```
 (x * y) ? b  =  (x ? b) * (y ? b)
 (x + y) ? b  =  (x ? b) + (y ? b)
(x + y) |+ a  =  (x |+ a) + (y |+ a)
(x |+ a) ? b  =  (x ? b) |+ a
```

Hertil kommer, at select spiller sammen med de logiske operatorer på følgende måde

```
x ? (b and c)  =  (x ? b) * (x ? c)
 x ? (b or c)  =  (x ? b) + (x ? c)
  x ? (not b)  =  x - (x ? b)
```

Der findes andre, mere subtile love, som vi ikke skal komme nærmere ind på her, men som sammen med ovenstående fx kan anvendes til effektivitetsfremmende omskrivninger.

Betragt som et simpelt eksempel herpå den distributive lov for union og project

```
(x + y) |+ a = (x |+ a) + (y |+ a)
```

Vi antager, at `x` og `y` hver indeholder ca. 10.000 tupler, men at `a`-attributten ikke indeholder mere end 100 forskellige værdier. Ved at udregne venstresiden skal der først skabes en relation `(x + y)` med ca. 20.000 tupler, som derefter skal projiceres ned til en relation med kun ca. 100 tupler. Ved en udregning af højresiden derimod nedskæres `x` og `y` udmiddelbart til beskeden størrelse, hvorefter foreningsmængden beregnes. Ved at bruge højresiden undgår vi altså at konstruere et unyttigt mellemresultat med ca. 20.000 tupler. Der findes andre eksempler, som viser at man kan opnå endnu mere dramatiske gevinster ved hjælp af passende omskrivninger af denne slags.

Til sidst betragter vi igen udtryk, hvor `+` og `*` angiver de sædvanlige aritmetiske operatorer. De færreste vil have problemer med at forstå følgende udtryk

```
x + y * z
```

selv om det igen kan opfattes på to forskellige måder, nemlig `x + (y * z)` og `(x + y) * z`. Når vi automatisk opfatter det første af disse alternativer som det korrekte, er det fordi vi ved, at `*` ifølge almindelig vedtægt *binder hårdere* end `+`. I RASMUS findes der ligeledes sådanne bindingsregler (eller *prioriteter*) for alle operatorer. I nedenstående prioriterede liste af operatorer binder de øverste hårdest og de nederste svagest.

```
not  |+  |-  !
and  *  /  mod
   or  +  -
     ?  :
=  <>  <  >  <=  >=
```

Dette betyder, at følgende (tvetydige) venstresider skal opfattes som angivet af de tilsvarende højresider

```
      x * y |+ a  =  x * (y |+ a)
       x + y ? b  =  (x + y) ? b
   b and c or d   =  (b and c) or d
     b or c < d   =  (b or c) < d
x or y = z and w  =  (x or y) = (z and w)
```

Det er ikke sikkert, at disse (og alle de øvrige) implikationer af operatorprioriteterne er i overensstemmelse med sædvanlig intuition. Det kan derfor normalt anbefales, at man sætter de tiltænkte paranteser eksplicit.

7 Opdateringer og Rapporter

I realistiske databaser er det ikke altid tilstrækkeligt at beregne nye relationer udfra gamle. Man skal også have mulighed for at opdatere tilstanden og for at udskrive rapporter i mere generelle formater.

7.1 Tilstandsændringer

I RASMUS kan man ændre den aktuelle tilstand ved hjælp af udtrykket

id := udtryk

Resultatet er det samme som for **udtryk**, men beregningen har den *sideeffekt*, at navnet **id** ændres til at have denne værdi. Når udtryk på denne måde kan forårsage tilstandsændringer, så giver det mening at udvide syntaksen med følgende konstruktion

$$\textbf{udtryk}_1 \ ; \ \textbf{udtryk}_2$$

der først beregner $\textbf{udtryk}_1$ og derefter $\textbf{udtryk}_2$, der giver det endelige resultat. Det første udtryk beregnes således kun fordi det har en sideeffekt.

Eksempel 32

Følgende funktion fjerner holdet `hold` fra `Runde1` og `Runde2`

```
Func (hold:Text) -> (Text)
  Runde1:=(Runde1?(#.HjHold<>hold) and (#.UdeHold<>hold));
  Runde2:=(Runde2?(#.HjHold<>hold) and (#.UdeHold<>hold));
  hold++" er nu fjernet"
end
```

7.2 Udskrifter

RASMUS understøtter en simpel mekanisme til at udskrive rapporter på tekstfiler. Udtrykket

`open(`**tekstudtryk**`)`

åbner UNIX-filen med navn **tekstudtryk**, der herefter er den aktuelle fil; udtrykket

`write(`**tekstudtryk**`)`

skriver **tekstudtryk** i den aktuelle fil; linjeskift angives med `"\n"`; udtrykket

`close`

lukker den aktuelle fil, så dens indhold bliver permanent.

Eksempel 33

Følgende udtryk skriver en lille historie på en fil

```
open("rødhætte");
write("Den lille Rødhætte gik ud i skoven,\n");
write("og så smed de ulven ned i brønden.\n");
close
```

7.3 Sorterede gennemløb

I forbindelse med relationer har vi flere gange understreget, at *rækkefølgen* af tuplerne ikke har nogen betydning. Det er dog ikke sandt for udskrifter, hvor man ofte gerne vil have oplysninger i fx alfabetisk orden. For at afhjælpe dette problem findes der to varianter af operatoren *forall* (og af *factor*). Udtrykkene

`!<(`**relation**`)`... og `!>(`**relation**`)`...

fungerer ligesom den almindelige *forall* eller *factor*, bortset fra, at tuplerne vælges sorteret efter de angivne attributter. Varianten `!<` sorterer i ikke-aftagende orden og varianten `!>` sorterer i ikke-voksende orden. For almindelige relationelle udtryk er en sådan rækkefølge ikke relevant, men når man genererer udskrifter, så kan den være meget nyttigt.

Eksempel 34

Dette udtryk udskriver på filen `holdliste` de deltagende fodboldhold i alfabetisk orden

```
open("holdliste");
write("De deltagende fodboldhold er:\n\n");
(!<(Kampe)|HjHold: (write(#.HjHold); write("\n")));
close
```

Opgave R1

Betragt følgende relationer

$$X = \begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 3 & 4 \\ \hline \end{array} \quad Y = \begin{array}{|c|c|}\hline A & B \\ \hline 1 & 3 \\ 5 & 7 \\ \hline \end{array} \quad Z = \begin{array}{|c|c|}\hline B & C \\ \hline 3 & 7 \\ 3 & 8 \\ 4 & 9 \\ \hline \end{array}$$

Beregn (med håndkraft) følgende udtryk

a) X `+` Y

b) X `-` Y

c) X `*` Z

d) Y `*` Z

e) X `*` Y

f) Z `|+` B

g) (Z `|+` B) `*` (Z `|+` C)

h) X `*` Y `*` Z

Opgave R2

Besvar følgende spørgsmål om materialet i `eksempler/fodbold90`.

a) På hvilke datoer blev der spillet uafgjort?

b) Hvilke hold har vundet alle deres hjemmekampe?

c) Hvilke hold har tabt til Brøndby?

d) Hvilke hold har vundet 2–1?

e) Hvilke par af hold har gensidigt vundet over hinanden?

Opgave R3

Hvilke af følgende udtryk er lovlige? Hvad er skemaerne af resultaterne af de lovlige udtryk?

a) `(Kampe * Resultater * Tips) |- HjHold`

b) `(Resultater + Runde2) - Runde1`

c) `Tips[Uge <- Dato] * Spilledag`

d) `Tips[Uge <- Dato] * ((Spilledag |- Dato)[Kamp <- Dato])`

Opgave R4

Forklar, hvad følgende udtryk beregner

```
(+ Val A = Runde1 + Runde2
   Val B = A * Resultater
   Val C = B ? (#.HjHold = "AGF")
   Val D = B ? (#.UdeHold = "AGF")
   Val E = C ? (#.HjScore > #.UdeScore)
   Val F = D ? (#.UdeScore > #.HjScore)
   Val G = E + F
   Val H = G[Id <- Kamp]
   Val I = H * Spilledag
   in I |+ Dato
+)
```

Opgave R5

Lav tipskuponen for uge 20 og 21 for materialet i `eksempler/fodbold90`. Dokumentér, hvordan du gjorde det. Skriv dem ud sorteret på `Nr`-attributterne. Beskriv, hvordan man givet en tipskupon og en *tipsrække* i form af en relation med skema

`Nr:Int`	`Tegn:Text`

kan beregne hvor mange rigtige tegn, der er på rækken.

Opgave R6

Beskriv, hvordan tilstanden ændrer sig under beregningen af følgende udtryk

```
(+ Val A = 3
   Val B = (+ Val C = A+2
              Val D = C*A
              in D-1
           +)
   Val C = (+ Val B = B+1
              Val E = (+ Val E = B-A
                         in B+E
                       +)
              in B+E
           +)
   in (+ Val D = A+B-C in 2*D +) + 3
 +)
```

Opgave R7

En *ruterelation* er af formen

fra	til
London	Paris
Tirstrup	London
Paris	New York
Paris	Moskva
London	Orlando

Den angiver de ruter, som et flyselskab tilbyder.

a) Skriv et udtryk, der beregner den *symmetriske lukning* af en ruterelation, det vil sige, sørger for, at hvis der er en rute fra A til B, så er der også en rute fra B til A.

b) Skriv et udtryk, der beregner den *refleksive lukning* af en ruterelation, det vil sige, sørger for, at der for alle byer A er en rute fra A til A (som man må formode er ret billig).

c) Lad `SAS` og `KLM` være to ruterelationer. Hvad beregner udtrykket

```
((SAS[til <- via])*(KLM[fra <- via])) |- via
```

Opgave R8

Skriv en funktion, der givet navnet på et hold beregner en relation med en enkelt attribut `Overmænd` af type `Text`, der indeholder navnene på de hold, der har besejret det angivne hold.

Opgave R9

Ved hjælp af *forall* og tupeludtryk kan man efterligne visse af de andre relationsoperatorer. Antag, at `X` er en relation med skema

a:Int	b:Int	c:Int	d:Text

Skriv *forall* udtryk, der giver samme resultat som

a) `X[a<-aa,c<-cc]`

b) `X |+ b,d`

Opgave R10

En attribut er en *nøgle*, hvis dens værdi entydigt bestemmer de øvrige attributter, eller med andre ord, hvis der ikke er to forskellige tupler med samme nøgleværdi. I fodboldmaterialet er fx `Id` og `Kamp` nøgler.

a) Forklar hvorledes udtrykket

```
not(Rel(Tup(x:true))<=(!(R):Rel(Tup(x:|Rel(Tup(A:#.A))*R|>1))))
```

afgør om attributten `A` er en nøgle i relationen `R`.

b) Kan du skrive et simplere udtryk med samme effekt?

Opgave R11

Det Cartesiske produkt (eller *krydsproduktet*) af to mængder A og B defineres som

$$A \times B = \{(a, b) \mid a \in A \ \wedge \ b \in B\}$$

For n mængder $A_1, \ldots, A_n$ definerer vi

$$A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) \,|\, a_1 \in A_1 \ \wedge \ldots \wedge a_n \in A_n\}$$

a) Lad $X = \{a, b\}$ og $Y = \{b, c, d, e\}$. Angiv $X \times Y$, $X \times X$ og $X \times X \times Y$.

b) Hvor mange elementer er der i $A_1 \times \cdots \times A_n$?

Potensmængden af en mængde A defineres som

$$\mathcal{P}(A) = \{S \,|\, S \subseteq A\}$$

det vil sige mængden af alle delmængder af A, inklusive $\{\}$ og A.

c) Lad $Z = \{a, b, c\}$. Angiv $\mathcal{P}(Z)$ og $\mathcal{P}(\{\})$.

d) Hvor mange elementer er der i $\mathcal{P}(A)$?

En *(matematisk) relation* over mængderne $A_1, \ldots, A_n$ defineres som et element i $\mathcal{P}(A_1 \times \cdots \times A_n)$. Hvad har dette med RASMUS relationer at gøre?

Opgave R12

En relation med følgende skema indeholder oplysninger om afleveringsopgaver på dProg1:

`Årskort:Int`	`Uge:Int`	`Godkendt:Bool`

For hver student og hver uge angives det, om den pågældende opgave er godkendt. Der skal skrives en funktion `OK` med hovede

```
Func (årskort:Int) -> (Bool)
```

der afgør, om den angivne students obligatoriske forløb er godkendt. Reglen er som bekendt, at mindst 8 opgaver skal godkendes, heraf skal mindst 5 være i ulige uger, og opgaven for uge 49 skal altid være godkendt.

Opgave R13

I `Spilledag` relationen skrives den 16. marts som `316` og den 1. april som `401`. Skriv en funktion `PænDato` med hovede

```
Func (dato:Text) -> (Text)
```

der ændrer formatet fra `"316"` til `"den 16. marts"`. Brug `PænDato` til at gøre `Spilledag` relationen pænere.

Opgave R14

En telefonbog kan repræsenteres som en relation med skema

Navn:Text	Nummer:Int

Skriv følgende seks funktioner; `Init` med hovede

```
Func () -> (Rel)
```

der giver en tom telefonbog; `Defined` med hovede

```
Func (TB:Rel, navn:Text) -> (Bool)
```

der fortæller, om `navn` forekommer i telefonbogen `TB`; `Dom` med hovede

```
Func (TB:Rel) -> (Rel)
```

der giver en relation med de navne, der forekommer i telefonbogen `TB`; `Lookup` med hovede

```
Func (TB:Rel, navn:Text) -> (Int)
```

der giver nummeret på `navn`, der skal findes i telefonbogen `TB`; `Update` med hovede

```
Func (TB:Rel, navn:Text, nummer:Int) -> (Rel)
```

der giver en udvidet kopi af telefonbogen `TB`, i hvilken `navn` har `nummer`; hvis `navn` allerede findes i `TB`, så skal nummeret blot ændres; endeligt er der funktionen `Delete` med hovede

```
Func (TB:Rel, navn:Text) -> (Rel)
```

der giver en kopi af telefonbogen `TB` i hvilken `navn` er fjernet.

Opgave R15

Skriv et udtryk, der ud fra en relation med skema

Hold:Text	Points:Int

(som på side 37) finder de tre bedste hold i turneringen.

Opgave R16

I `eksempler/astronomi` findes en relation `planeter` med skema

Planet	Navn	Afstand	Diameter	Opdaget

der indeholder oplysninger om planeter, deres måner, månens afstand fra planeten, månens diameter, og årstallet hvor månen blev opdaget (af mennesker). Skriv og udfør RASMUS udtryk, der beregner

a) en relation med navnene på de planeter, der har større måner end vores.

b) en relation med navnene på de planeter, der i år 1850 ikke havde kendte måner.

c) en relation med samme skema som `planeter`, men som for hver planet kun indeholder tuplet med dennes største måne.

Opgave R17

I `eksempler/benzin` ligger en relation `forbrug` med skema

Dato:Text	Km:Int	Volumen:Int	Pris:Int

der angiver benzinforbruget for en bil (med en pedantisk ejer). Hvert tupel beskriver en optankning med dato, bilens kilometertæller, volumen af indkøbt benzin (i centiliter) og prisen (i ører). Skriv RASMUS udtryk, der beregner

a) den gennemsnitlige pris betalt pr. liter benzin i perioden.

b) den dato, hvor benzinen var dyrest.

c) en funktion, der givet et årstal (som tekst: `"88"`, `"89"`, etc.) beregner, hvor mange km bilen har kørt i dette år.

d) hvor sikker ejeren er på hånden; det vil i denne sammenhæng sige, hvor tæt han i gennemsnit er på at tanke et helt antal liter.

Opgave R18

Betragt igen ruterelationerne fra opgave R7. Med *produktet* `R·S` mener vi resultatet af

```
((R[til <- via])*(S[fra <- via])) |- via
```

Vi forkorter $\texttt{R}^1 = \texttt{R}$ og $\texttt{R}^n = \texttt{R}{\cdot}\texttt{R}^{n-1}$ for $n > 1$. Den *transitive lukning* af `R` defineres som

$$\texttt{R}^+ = \sum_{i=1}^{\infty} \texttt{R}^i = \texttt{R}{+}\texttt{R}^2{+}\texttt{R}^3{+}\texttt{R}^4{+}\ldots$$

a) Hvilke oplysninger om ruter indeholder $\texttt{R}^+$?

b) Argumentér for, at der for alle `R` findes et $n \geq 1$, sådan at

$$\texttt{R}^+ = \sum_{i=1}^{n} \texttt{R}^i$$

Betragt følgende udtryk

```
(+ Val Prod  = func (S,R: Rel) -> (Rel)
                 ((S[til <- via])*(R[fra <- via])) |- via
               end
   Val Trans = func (X: Rel) -> (Rel)
                 (+ Val P = Prod(X,R)+R
                    in if P=X -> X
                        & true -> Trans(P)
                       fi
                 +)
               end
   in Trans(R)
+)
```

c) Argumentér for, at udtrykket beregner $\texttt{R}^+$.

Opgave R19

Betragt en relation `Tilmeldinger` med skema

Årskort:Text	Kursus:Text

der angiver, hvilke studenter, der ønsker at følge hvilke kurser. Til brug for skemaplanlægning ønsker vi en relation `Konflikter` med skema

Kursus1:Text	Kursus2:Text	Antal:Int

hvor tuplerne angiver, at blandt de tilmeldte til `Kursus1` er der `Antal` studenter, der også har tilmeldt sig `Kursus2`. Skriv et udtryk, der foretager denne beregning (vink: factor).

Opgave R20

Afgør hvilke af følgende algebraiske love, der gælder for RASMUS udtryk. Giv en *kort* begrundelse for hver lov, der gælder, og angiv et modeksempel for hver lov, der *ikke* gælder. Som i noten kan det antages, at begge sider af ligningerne er lovlige udtryk.

a) `(x * y) |+ a = (x |+ a) * (y |+ a)`

b) `(x * x) = x`

c) `(x |+ a) * (x |- a) = x`

d) `(x ? b) + (x ? not b) = x`

e) `x - (y - z) = (x - y) + z`

f) `(x |- a) - (y |- a) = (x - y) |- a`

g) `x ? (a and not b) = (x ? a) - (x ? b)`

Opgave R21

Modificér opgave R5, således at tipskuponen udskrives i det velkendte format på en tekstfil.

Indeks