PB - 215

J.L. Knudsen: Programmeringssprog — Begreber og undtagelser

TRYK: RECAU (06) 12 83 55

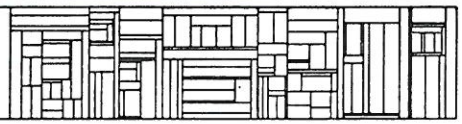
ISSN 0105-8517

Programmeringssprog: Begreber og undtagelser (Resumé)

Jørgen Lindskov Knudsen

DAIMI PB - 215
Juni 1986

DATALOGISK AFDELING
Bygning 540 - Ny Munkegade - 8000 Aarhus C
tlf. (06) 12 83 55, telex 64767 aausci dk
Matematisk Institut Aarhus Universitet



Programmeringssprog: Begreber og undtagelser (Resumé)

Jørgen Lindskov Knudsen
Datalogisk afdeling
Aarhus Universitet

Dette notat er en sammenfattende redegørelse for de arbejder, der er indleveret med henblik på erhvervelse af den naturvidenskabelige licentiatgrad ved Aarhus universitet. Licentiatstudiet er foretaget ved Datalogisk afdeling, Matematisk institut, Aarhus universitet med lektor Ole Lehrmann Madsen som faglig vejleder.

Notatet indeholder en almen beskrivelse af den behandlede problemstilling, samt en motivation for angrebsvinklen på denne problemstilling. Herudover gives en redegørelse for de opnåede resultater, samt for relationerne til den øvrige forskning indenfor området.

Studiet af programmeringssprog foretages udfra en lang række forskellige synsvinkler. Det er vigtigt at gøre sig klart, at ingen af disse synsvinkler eksisterer i total isolation. Der foregår til stadighed en udveksling af ideer, begreber og metoder imellem disse synsvinkler. Jeg vil i det følgende præsentere de vigtigste synsvinkler. Denne præsentation vil samtidigt give et indblik i bredden i forskningen indenfor emneområdet.

Implementation of programmeringssprog

Indenfor dette delområde beskæftiger man sig med emner som grammatikker, parsning af tekster udfra en given grammatik, syntaktisk og semantisk analyse, kompilering og fortolkning, samt kodegenerering og kodeoptimering. I de seneste år er dette delområde blevet udvidet til at omfatte programmeringssystemer. Programmeringssystemer er programsystemer, der, udover at stille en implementation af et programmeringssprog til rådighed, indeholder en lang række hjælpeværktøjer, der understøtter programmørens arbejde. Disse systemer omtales ofte populært som »programmørens høvlebænk«. Indenfor dette delområde er de vigtigste emner interaktion, inkrementalitet, uniform repræsentation, integration, samt multiple eksterne repræsentationer.

Det er oplagt, at ovennævnte synsvinkel er den ultimative synsvinkel, der må lægges på programmeringssprog for at realisere disse som brugbare værktøjer. Til støtte for denne synsvinkel findes i det væsentligste to andre synsvinkler: *Formel beskrivelse af programmeringssprog* samt *Design af programmeringssprog*.

Formel beskrivelse af programmeringssprog

Formel beskrivelse af programmeringssprog forskes i to forskellige retninger: syntaktisk beskrivelse af programmeringssprog og semantisk beskrivelse af programmeringssprog.

Syntaktisk beskrivelse af programmeringssprog fokuserer på syntaksen for den notation, der defineres af programmeringssproget, og beskæftiger sig bla. med formelle grammatikker (eks. Chomsky grammatikker, Attribut grammatiker, W-grammatikker), parsemetoder og abstrakt syntaksrepræsentation. Dette område er for øjeblikket ikke genstand for stor forskningsaktivitet.

Semantisk beskrivelse af programmeringssprog fokuserer på betydningen af den notation, der defineres af programmeringssproget, og beskæftiger sig med formelle modeller for beregninger. Af sådanne modeller bør nævnes axiomatisk semantik, operationel semantik, denotationel semantik, aktionssemantik, CCS, samt Petri-net.

Design af programmeringssprog

Denne synsvinkel har været den mest dominerende under dette licentiatstudium og afstikker som sådan de ydre rammer for studiet. Design af programmeringssprog kan antage to former: Design af sproglige udtryksmidler til løsning af specifikke opgaver indenfor programmeringssprog, samt design af et sammenhængende programmeringssprogsforslag. Disse to former stiller forskellige krav til udøveren. Design af sproglige udtryksmidler kræver en fokusering på et specifikt problem (eller et specifikt problemkompleks) indenfor eksisterende programmeringssprog, hvorimod design af et sammenhængende programmeringssprogforslag kræver et indgående kendskab til alternative sproglige udtryksmidler. Valget mellem alternativerne bør foretages på baggrund af en overordnet idé/filosofi, og de valgte sproglige udtryksmidler må derefter tilpasses hinanden, således at de tilsammen fremstår som et uniformt design og ikke som en samling relativt enkeltstående udtryksmidler.

Design af programmeringssprog foregår primært indenfor rammerne af en *programmeringsstil*. Der er i det væsentligste fire forskellige programmeringsstile: Algoritmisk programmering, funktionel programmering, logik programmering, samt objekt-orienteret programmering.

Algoritmisk programmering tager sit udgangspunkt i *algoritmer*, der ændrer *tilstanden* af et *lager* i en datamaskine. Lageret repræsenteres i programmeringssproget ved variabelbegrebet, og algoritmerne ved program- og procedure-begreberne. Traditionelt har variable typer, der abstrakt specificerer, hvilke tilstande, den del af lageret, variablen repræsenterer, kan antage. Procedurerne er imperative (angiver tilstandsovergange ved trinvis modifikation af tilstanden af variable), og der benyttes elaborerede kontrolstrukturer til at angive rækkefølgen af de enkelte tilstandsovergange.

Programmeringssprogene Cobol, Fortran, Algol60, Pascal, Modula-2, Ada er eksempler på programmeringssprog, der primært er rettet mod understøttelse af den algoritmiske programmeringsstil.

Funktionel programmering tager sig udgangspunkt i *værdier* samt *funktioner*. Værdierne er universelle, tidsløse abstraktioner, og funktionerne tager værdier som argumenter og leverer andre værdier som resultat. Da værdier er tidsløse, bliver funktionerne det ligeledes. Kontrolaspektet i funktionel programmering er dækket af betingede udtryk, funktionskald, samt rekursion.

Funktionel programmering er således baseret på matematisk teori og en lang række matematiske regler kan finde anvendelse indenfor denne programmeringsstil.

Programmeringssprogene Lisp og ML er eksempler på programmeringssprog, der primært er rettet mod understøttelse af den funktionelle programmeringsstil.

Logik programmering tager sit udgangspunkt i *konstanter*, *relationer* samt *deduktionsregler*. Konstanter er navne, der repræsenterer specifikke objekter. *Objekterne* i et logikprogram har ingen betydning i programmet, men relateres til objekter i det system, som man ønsker at modellere. Dvs. fortolkningen af objekterne ligger udenfor logikprogrammet. Relationerne repræsenterer sammenhænge mellem konstanter. Konstanter og relationer kaldes samlet for *fakta*, idet de i et logikprogram repræsenterer den viden, der er explicit repræsenteret i programmet. Deduktionsreglerne repræsenterer derimod metaviden, dvs. viden om viden. En deduktionsregel specificerer eksempelvis, at hvis en given relation gælder mellem et antal konstanter, så gælder en anden relation mellem de samme konstanter.

Kontrolaspektet er implicit i logik programmering, idet kontrollen defineres af den underliggende inferens-maskine. Af effektivitetshensyn findes dog forskellige sproglige udtryksmidler til angivelse af kontrol-information til inferens-maskinen (eks. *cut* og *fail* i Prolog).

Logik-programmering er baseret på 1.ordens prædikat logik. 1.ordens prædikat logik er dog for generelt til direkte anvendelse som programmeringssprog til understøttelse af logik programmering, og der anvendes derfor sædvanligvis en restringeret logik, baseret på Horn-udtryk.

Programmeringssprogene Prolog, C-Prolog, Parlog, Scanlog er eksempler på programmeringssprog, der alle primært er rettet mod understøttelse af logik programmering.

Objekt-orienteret programmering tager sit udgangspunkt i *objekter* og *klasser*. Objekter har tilstand og dermed selvstændig eksistens, og objekter kan udføre handlinger. Objekternes handlinger kan enten ændre deres egen lokale tilstand, eller de kan anmode andre objekter om at udføre specifikke handlinger. Objekter har egenskaber/attributter. Disse egenskaber/attributter kan bla. angive handlinger, som objektet accepterer at blive anmodet om at udføre. Objekter er instantieret udfra deskriptorer. Deskriptorerne

beskriver, hvilke tilstande et objekt kan antage; hvilke handlinger, det kan udføre; samt hvilke handlinger, det vil kunne acceptere at blive anmodet om at udføre. Deskriptorerne kan være organiseret i et klasse-hierarki (sædvanligvis træ-struktureret). Klasse-hierarkiet anvendes til at specificere arvelighed mellem klasserne i hierarkiet. Hvis vi her indskrænker os til at betragte træ-strukturerede klasse-hierarkier, så kan vi forklare arveligheden mellem klasserne således, at en klasse arver fra en anden klasse, hvis der findes en vej i hierarkiet mellem de to klasser. De klasser, der er nærmest roden i hierarkiet, er de mest generelle klasser. De klasser, der er længere nede i hierarkiet, er de mest specielle. Vi siger, at de specielle klasser arver fra de generelle klasser (arvelighed går altså fra roden ud mod bladene i hierarkiet). Arvelighed skal her forståes således, at klasser kun beskriver objekternes egenskaber/attributter partielt. Dette betyder, at en instans af en klasse besidder de egenskaber/attributter, der er beskrevet i klassen selv, samt alle de egenskaber, som klassen arver fra de mere generelle klasser. Denne arvelighed kan benyttes til at erstatte det traditionelle type-begreb med et kvalifikationsbegreb. Kvalifikationsbegrebet benytter klassehierarkiet til at tillade, at et objekt manipuleres i overensstemmelse med en anden klasse end den klasse, den er en instans af, hvis den instantierende klasse arver fra denne anden klasse.

Programmeringssprogene Simula67, Smalltalk-80, Beta, Loops er eksempler på programmeringssprog, der alle primært er rettet mod at understøtte objekt-orienteret programmering.

Den objekt-orienterede programmeringsstil har dannet den programmeringsstilistiske baggrund for mit licentiatstudium.

Begrebsmæssig angrebsvinkel på programmeringssprog

En del af licentiatstudiet har angrebet programmeringssprogområdet vha. en begrebsmæssig angrebsvinkel. Den begrebsmæssige angrebsvinkel indeholder analyse og opbygning af begreber, der er uafhængige af det enkelte programmeringssprog. Begreber, der er velegnede til brug for analyse af programmeringssprog, således at denne analyse ikke kommer til at foregå på det enkelte sprogs egne præmisser, men udfra et alment, programmeringssprogligt begrebsapparat.

Dette arbejde er beskrevet i *»A Conceptual Framework for Programming Languages«* (ref./1/), som er udarbejdet i fællesskab med Kristine Stougård Thomsen. Rapporten er ligeledes indsendt af førnævnte, som en del af hendes licentiatafhandling mhp. erhvervelsen af den naturvidenskabelige licentiatgrad ved Aarhus universitet.

»A Conceptual Framework for Programming Languages« er en analyse af programmeringsprocessen mhp. at afklare programmeringssproglig understøttelse af programmeringsprocessen. Analysen foretages på baggrund af og giver anledning til dannelsen af en sproguafhængig begrebsramme for diskussion af programmeringssprog og programmeringssproglige udtryksmidler. Denne begrebsramme afspejler to fundamentale opfattelser af programmeringssprog. Den første opfattelse afspejler programmeringssproget som et værktøj, som skal benyttes til at styre datamaskinens beregninger/operationer. Denne opfattelse giver anledning til opstillingen af en model for program-udførelser — *deskriptor-entitetsmodellen*. Den anden opfattelse afspejler programmeringssproget som et værktøj til begrebsmodellering. Herved forståes, at programmeringssproget skal benyttes til at specificere en model i datamaskinen — en model, der afspejler begreber og fænomener i brugerens begrebsverden. Denne begrebsmæssige angrebsvinkel giver anledning til en analyse af begrebsstrukturer med særlig vægt på abstraktionsteknikker.

Artiklen er opdelt i fire kapitler. Kapitel 1 karakteriserer programmeringsprocessen og

leder frem til de to ovennævnte opfattelser af programmeringssprog. Kapitlet bygger i det væsentligste på forskning indenfor systembeskrivelse.

Kapitel 2 analyserer begrebsforståelse og abstraktion for at opnå en forståelse af begrebsmodellering. Denne analyse bygger i det væsentligste på forskning indenfor databaseområdet, kunstig intelligens og grænseområdet herimellem.

Kapitel 3 analyserer fundamentale elementer af programmeringssprog udfra synsvinklen om programmeringssprog som styringsværktøj. Heri opbygges *deskriptor-entitets*modellen for programudførelser. Kapitlet bygger i det væsentligste på forskning indenfor Beta-projektet.

Kapitel 4 repræsenterer syntesen af kapitel 2 og 3. Kapitlet giver en detaljeret analyse af programmeringssporglig understøttelse af de forskellige abstraktionsteknikker, diskuteret i kapitel 2, på baggrund af *deskriptor-entitets*modellen for programudførelser, diskuteret i kapitel 3. Kapitel 4 er således kernen i artiklen, og udgør syntesen af vores arbejde med den begrebsmæssige angrebsvinkel til programmeringssprog.

Sproglige udtryksmidler til undtagelseshåndtering

Udover ovenstående arbejder, hvor angrebsvinklen er en begrebsmæssig og analyserende angrebsvinkel, har licentiatstudiet indeholdt en konstruerende angrebsvinkel, idet jeg har arbejdet med forslag til programmeringssproglige udtryksmidler til specifikation af kontrol, herunder specielt til specifikation af usædvanlige kontrol-veje i et program (sædvanligvis kaldet »exception handling« eller »undtagelseshåndtering«). Dette arbejde er beskrevet i referencerne /2/, /3/ og /4/.

Motivationen for dette arbejde er, at de eksisterende sproglige udtryksmidler til undtagelseshåndtering ofte fremstår som fremmede elementer i deres værtssprog — fremmed elementer med helt egne regler, som kan være i modstrid med værtssprogets overordnede regelsæt. Som eksempel kan nævnes programmeringssproget Ada, der er et sprog med statisk navnebinding som en overordnet regel. Ada's sproglige udtryksmiddel til undtagelseshåndtering benytter derimod en dynamisk bindingsregel.

Inden jeg går over til at beskrive indholdet af referencerne /2/, /3/ og /4/, vil jeg give en kortfattet beskrivelse af emnekredsen undtagelseshåndtering.

Undtagelseshåndtering

Undtagelseshåndtering har traditionelt omhandlet problemstillinger omkring opståede fejlsituationer, men er i de senere år blevet drejet i retning af at dække beregninger, bestående af et hovedforløb (»normalberegningen«) samt et eller flere underordnede forløb (»specialberegninger«). Bemærk, at jeg her anvender ordet »beregning« i en meget bred betydning, dækkende over såvel numeriske beregninger, symbolske manipulationer, som simuleringer, etc.

Mit arbejde indenfor dette område har været koncentreret omkring programmeringssprog med blokstruktur samt statisk navnebinding. De forslag til sproglige udtryksmidler til undtagelseshåndtering, der er stillet indenfor disse sprog, benytter alle en dynamisk navnebindingsregel, når navnet på en undtagelse skal associeres med en handler. Dette sker, når en undtagelse rejses under normalberegningen (normalt vha. et imperativ, der navngiver den opståede undtagelse). Når en undtagelse rejses, skal en specialberegning initieres (en handler for en undtagelse specificerer denne specialberegning). Samtlige nyere forslag er enige om, at hovedberegningen skal afbrydes, når en undtagelse rejses. Mao. en

forekomst af en undtagelse betyder, at den resterende del af hovedberegningen erstattes af en specialberegning.

I de føromtalte blokstrukturerede programmeringssprog med statisk navnebinding er et forslag til sproglige udtryksmidler til undtagelseshåndtering, der benytter en dynamisk navnebindingsregel, i modstrid til sprogets generelle struktur. Jeg har derfor undersøgt muligheden af at designe et sprogligt udtryksmiddel til undtagelseshåndtering, der benytter sig af en statisk navnebindingsregel.

Sequels anvendt til undtagelseshåndtering

Motivationen til arbejdet er ovennævnte misforhold mellem de navnebindingsregler, der benyttes til undtagelseshåndtering og sprogenes generelle navnebindingsregler. Inspirationen til udformningen af det sproglige udtryksmiddel skal findes i R.D. Tennents artikel: »*Language Design Methods based on Semantic Principles*«, Acta Informatica, 8(2), pp.97-112 (1977), hvori han introducerer en generalisering af det velkendte **goto**-begreb i retning af procedure-begrebet. Dette sproglige udtryksmiddel kaldes en **sequel**.

Som en illustration af dette udtryksmiddel vil vi betragte eksempel 1.

```
function TreeLookup( t: Tree, k: Key, sequel NotFound ): Item;
  sequel Found( i: Item ) begin TreeLookup:= i end;
  procedure Traverse( t: Tree);
  begin
    if t <> nilTree then
      if t.key = k then Found( t.item )
      else begin
             Traverse( t.left );
             Traverse( t.right );
           end
  end;
begin Traverse( t ); NotFound end;
```

Eksempel 1: Tree lookup

Bemærk, at sequel-begrebet ikke er anvendt til at håndtere en fejlsituation, men derimod til at angive den undtagelse, det er at finde (hhv. ikke finde) det ønskede element under et gennemløb af træet. Betydningen af de to sequels *Found* og *NotFound* kan forklares ved at betragte eksempel 2, hvori *TreeLookup* benyttes. Lad os betragte de to mulige situationer:

- '*' er i træet *ExpTree*.
- '*' er ikke i *ExpTree*.

I det første tilfælde vil et af de rekursive kald af *Traverse* have *t.key* = '*' og sequel *Found* vil blive kaldt. Dette vil resultere i udførelsen af kroppen af *Found*, hvorefter kaldet *TreeLookup(ExpTree, '*', LookupError)* vil terminere, returnerende værdien af *t.item*, som derefter vil blive tilordnet *Val*. Kaldet af *Found* indikerede altså den undtagelse, at '*' fandtes under gennemløbet af *ExpTree*, og normalberegningen (i dette tilfælde gennemløbet af træet) afbrydes og erstattes af den (i dette tilfælde simple) specialberegning, der består af klargøring af returværdien af *TreeLookup*.

```
declare
  function TreeLookup(...): Item;
    begin ... end;
begin
  :
  declare
    Exptree : Tree;
    Val : Item;
    sequel LookupError;
      begin
        (* do some actions in case *)
        (* of a lookup error *)
      end;
  begin
    :
    Val:= TreeLookup( ExpTree, '*', LookupError );
    :
  end;
  :
end;
```

Eksempel 2: TreeLookup anvendelse

I det andet tilfælde, hvor '*' ikke er i *ExpTree*, vil *Traverse* gennemløbe hele *ExpTree* og returnere til *TreeLookup*, der ved kaldet af *NotFound* indikerer, at i forhold til *TreeLookup* er dette en fejl. Konsekvensen af kaldet af *NotFound* er, at kaldet af *TreeLookup(ExpTree, '*', LookupError)* afbrydes, efter at kroppen af *LookupError* er blevet udført (*LookupError* er jo aktuel parameter for *NotFound*). Samtidigt afbrydes hele blokken i eksempel 2.

Dette eksempel illustrerer, at en forekomst af en specifik undtagelse i et program kan betragtes på tre niveauer. Niveau 1 er den del af programmet, hvor en forekomst af en specifik undtagelse betragtes som en fejl. I eksempel 2 er niveau 1 for *Found* hele kroppen af *Traverse*. Mao. i *Traverse* betragtes dette at finde elementet for en fejl.

Niveau 2 er den del af programmet, hvor en forekomst af en specifik undtagelse betragtes som en usædvanlig men håndterbar situation. I eksempel 2 er niveau 2 for *Found* hele kroppen af *TreeLookup*. Mao. i *TreeLookup* betragtes dette at finde elementet blot som en håndterbar situation.

Niveau 3 er den del af programmet, hvor en forekomst af en specifik undtagelse er usynlig eller ikke eksisterende. I eksempel 2 er niveau 3 for *Found* hele eksemplet undtagen kroppen af *TreeLookup*. Mao. selv om *Found* er en undtagelse i *TreeLookup*, er undtagelsen kapslet ind, således at forekomster af den ikke kan registreres udenfor *TreeLookup*.

Diskussion

Et argument imod specielle mekanismer til undtagelseshåndtering er ofte, at man bør teste for undtagelser, inden en beregning initieres. Der er dog en lang række tilfælde, hvor dette argument ikke holder. For det første kan det være ret kostbart (målt i køretid) at teste for undtagelsessituationer før beregningen foretages. Ovenikøbet er det ofte således, at undtagelsestestet kan foretages som en integreret del af hovedberegningen, hvorved undtagelsestestet ikke giver et væsentligt overhead.

For det andet er der situationer, hvor det er umuligt at foretage testet på forhånd. Et eksempel herpå er input-rutiner. Her er et forhåndstest åbenlyst umuligt, og specielle mekanismer er derfor bydende nødvendige.

Statisk undtagelseshåndtering

Mit arbejde med statisk undtagelseshåndtering er beskrevet i referencerne /2/, /3/ og /4/. I artiklen *»Exception Handling — A Static Approach«* (ref./2/) beskrives sequel-begrebet, og der gives eksempler på anvendelser. Herudover introduceres et nyt sprogligt udtryksmiddel, *afledt definition*, der gør det muligt at binde parametre til parametriserede programenheder. Afledt definition er inspireret af programmeringssproget Ada, og giver mulighed for at lave specialiserede programenheder udfra mere generelle programenheder. Afledt definition viser sig specielt velegnet i forbindelse med sequels som parametre til programenheder, idet anvendelsen af afledt definition på en sådan programenhed, hvor den afledte definition binder en eller flere sequel-parameter, giver en meget flexibel notation. Herudover diskuteres, hvordan den predefinerede del af et sprog bør designes for at kunne understøttes af et statisk undtagelsesbegreb som det foreslåede sequel-begreb. Endelig diskuteres en række af de øvrige forslag til sproglige udtryksmidler til undtagelseshåndtering.

Det skal til sidst bemærkes, at afsnit 2.9 i reference /2/ diskuterer en ændret semantik af sequels i forhold til den semantik, der gives i R.D. Tennents førnævnte artikel. Lektor J. Steensgaard-Madsen, DtH, har senere påpeget, at der fandtes tilfælde, hvor denne ændrede semantik ikke levede op til de krav, der blev stillet i afsnittet. Efter at have studeret denne problematik igen, blev to ting klart. For det første, at det *er* muligt at give en semantik, der generelt lever op til de krav, der blev stillet op i førnævnte afsnit — men denne semantik er kompliceret. For det andet, viste det sig, at selvom en semantik kan gives, er det ikke oplagt, at den bør bringes i anvendelse, idet de krav, der blev stillet i afsnittet, kan der stilles spørgsmål ved. Jeg vil i det følgende argumentere herfor.

Lidt simplificeret går kravene i førnævnte afsnit ud på, at hvis undtagelse **A** forekommer under behandlingen af undtagelse **B**, så må undtagelse **A** ikke influere på termineringsfølgerne af behandlingen af undtagelse **B**. Jeg vil gerne med eksempel 3 illustrere, at dette krav ikke er oplagt. Eksemplet viser en blok, hvori bla. en fil ved navn *f* ønskes åbnet. Hvis filen eksisterer, skal den naturligvis blot åbnes. Hvis filen ikke eksisterer, skal det forsøges at skabe en ny fil i det nuværende filkatalog. Hvis dette ikke er muligt, skal dette betragtes som en usædvanlig hændelse. Dette er i eksempel 3 modelleret på flg. måde: I *outer-block* er en sequel *FileNotFound* erklæret med en anden sequel *Recover* som formel parameter. *FileNotFound* forsøger at skabe en ny fil, og hvis dette lykkedes, kaldes sequel *Recover*; ellers foretages normal undtagelseshåndtering for *outer-block*. I handlingsdelen for *outer-block* foretages åbningen af filen ved navn *f* i en indre blok *inner-block*. I *inner-block* er erklæret en sequel *FileCreated*, der åbner filen og foretager fil-initialiseringer. I handlingsdelen for *inner-block* testes, hvorvidt filen eksisterer eller ej. Hvis filen ikke eksisterer, betragtes det umiddelbart som en fejlsituation, og sequel *FileNotFound* kaldes med *FileCreated* som aktuel parameter. Hvis det ikke lykkedes at skabe en ny fil ved navn *f*, vil hele *outer-block* terminere efter normal undtagelseshåndtering. Hvis det derimod lykkedes for *FileNotFound* at skabe en ny fil ved navn *f*, vil *FileCreated* blive kaldt, hvorved filen åbnes og initieres og endelig *inner-block* termineres.

I dette eksempel ville det mest logiske være at fortsætte efter det terminerende **end** af *inner-block*, idet filen er åben og klar til videre behandling. Dette ville altså betyde, at den oprindelige terminering af *outer-block* (indikeret af kaldet af *FileNotFound*) ikke bliver håndhævet. Diskussionen i afsnit 2.9 i reference /2/ omhandler netop denne situation,

```
declare (* outer-block *)
  f : FileName;
  sequel FileNotFound( sequel Recover );
    begin
      if CreatePermission
      then begin Create(f); Recover end
      else (* do normal exception handling *)
    end;
begin (* outer-block *)
  :
  declare (* inner-block *)
    sequel FileCreated
      begin Open(f); (* init file: f *) end;
  begin (* inner-block *)
    if FileNonExistent(f)
    then FileNotFound(FileCreated)
    else Open(f);
  end;
  :
end; (* outer-block *)
```

Eksempel 3: Undtagelseshåndtering afbrudt af en anden undtagelse

som abstrakt set er uhensigtsmæssig. Eksempel 3 viser dog, at evnen til at annulere en terminering giver en øget flexibilitet, der er meget anvendelig i praksis. Da dette giver en langt simplere semantik (og ligeledes en simplere implementation), er den oprindelige semantik for sequels anvendt som basis for det videre arbejde i referencerne /3/ og /4/. Afsnit 2.9 i reference /2/ skal derfor betragtes som et skridt i den forkerte retning og derfor annuleres.

Statisk undtagelseshåndtering i blok-strukturerede systemer

I artiklen *»A Hierarchical, Co-operative Exception Handling Mechanism«* (ref./3/) anvendes sproglige udtryksmidler stammende fra objekt-orienteret programmering på sequel-begrebet. Motivationen herfor er to forhold. For det første har de dynamiske forslag muligheder for at specificere undtagelseshandlere for den samme undtagelse, således at en forekomst af en undtagelse, udover at terminere en række lag i programmet, kan specificere oprydning af hvert enkelt lag i programmellet. For det andet var jeg interesseret i at undersøge, hvorvidt statisk undtagelseshåndtering og objekt-orienteret programmering kan tilpasses hinanden.

De sproglige udtryksmidler til understøttelse af objekt-orienteret programmering, som jeg har fokuseret på, er prefixing og virtuel binding. Jeg vil ikke give en generel introduktion til disse udtryksmidler i dette notat, men henvise til appendiks A og B i reference /3/, samt kapitel 4 i reference /1/, hvori disse udtryksmidler beskrives og diskuteres i detaljer. I stedet vil jeg her give et eksempel på, hvordan disse udtryksmidler bidrager til udvidelsen af udtryksstyrken af sequel-begrebet.

I dette notat er det på sin plads at bemærke, at inspirationen fra objekt-orienteret programmering kunne have markeret sig på en anden måde i reference /3/ (og senere i reference /4/). Jeg kunne have valgt at designe nye sproglige udtryksmidler, der adskildte sig fra prefixing og virtuel binding, men samtidigt bibeholdt målet om lagdelt terminering.

Dette ville have drejet denne diskussion væk fra objekt-orienteret programmering og blot bidraget til den mangfoldighed af forslag til sproglige udtryksmidler, der eksisterer i dag. Jeg har ikke ønsket denne udvikling. Tværtimod har jeg stræbt imod at forene eksisterende udtryksmidler og påvise styrken i denne konstellation. Samtidigt ønskede jeg at bidrage til at introducere statisk undtagelseshåndtering i objekt-orienteret programmering.

»A Hierarchical, Co-operative Exception Handling Mechanism« (ref./3/) beskæftiger sig med ovennævnte indenfor rammerne af et traditionelt blok-struktureret programmeringssprog (såsom Pascal eller Ada). Årsagen hertil er at afprøve ideernes holdbarhed uden at introducere ekstra komplexitet (reference /4/ beskæftiger sig med statisk undtagelseshåndtering indenfor rammerne af et objekt-orienteret, blok-struktureret programmeringssprog). I artiklen præsenteres begrebet lagdelte systemer, og problemstillingen omkring undtagelseshåndtering i sådanne lagdelte systemer diskuteres. Udfra denne diskussion præsenteres sequels med prefixing og virtuel binding som forslag til et system af sproglige udtryksmidler til undtagelseshåndtering, der tilsammen udgør et redskab til undtagelseshåndtering i lagdelte systemer.

Lad os som et eksempel på anvendelsen af ovennævnte betragte eksempel 4. Eksemplet er en skabelon for et tænkt simuleringsprogram, hvori simuleringen består af et antal iterationer, hvor man efter hver iteration kan afgøre, hvorvidt simuleringen skal fortsættes; hvorvidt sidste iteration skal annuleres; hvorvidt simuleringen skal afsluttes; eller endelig hvorvidt simuleringen skal annuleres. I eksemplet er erklæret syv sequels, hvoraf de tre er prefixede. Eksemplet illustrerer ikke brugen af virtuel binding — der henvises til eksempler i reference /3/ og /4/.

Eksemplet består af fire indlejrede blokke. Den yderste blok *simulation-block* er den blok, hvori selve simuleringen foretages. Simuleringen kan bringes til øjeblikkelig standsning ved kald af sequel *AbortSimulation*. Blokken *loop-block* er den ydre løkke i simuleringen. Simuleringen kan afsluttes ved kald af sequel *TerminateIteration*, der vil terminere *loop-block*. Hver iteration består af udførelse af blokken *inner-loop-block*. Sequel *UndoLastIteration* i *inner-loop-block* terminerer *inner-loop-block* efter at have annuleret det igangværende iterationsskridt. Inde i *inner-loop-block* implementerer blokken *inner-block* det beslutningspunkt, hvor simuleringens videre forløb kan bestemmes af brugeren. Valget mellem de forskellige muligheder foretages ved, at en menu (*Prompter*) vises på skærmen, og brugeren kan så vælge mellem de forskellige muligheder ved at udvælge et felt i menuen. Hvert felt i menuen er koblet til en sequel, således at valget udløser et kald af den tilhørende sequel.

Lad mig kommentere de fire sequels, der er koblet til hver af de fire felter. Alle fire er erklæret i den inderste blok i eksemplet.

Continue er ikke prefixet med andre sequels og vil derfor terminere den inderste blok, efter at have fjernet menuen fra skærmen. Herefter vil simuleringen fortsætte med forberedelserne til næste iteration.

Undo er prefixet med *UndoLastIteration*, der er erklæret i blokken *inner-loop-block*. *UndoLastIteration* er ikke prefixet med andre sequels, og *Undo* vil derfor terminere blokken *inner-loop-block* efter at have udført først *ClosePrompter* og derefter *(*and undo last iteration*)*. Herefter vil simuleringen starte en ny iteration, som om den seneste iteration ikke havde været foretaget.

Terminate er prefixet med *TerminateIteration*, der er erklæret i tilknytning til løkken (blokken *loop-block*). *TerminateIteration* er ikke prefixet med andre sequels, og *Terminate* vil derfor terminere *loop-block* efter at have udført først *ClosePrompter*,

```
declare (* simulation-block *)
  :
  sequel AbortSimulation;
    begin (* close simulation window *) end;
begin (* simulation-block *)
  (* initiate the simulation and setup simulation window *)
  :
  declare (* loop-block *)
    sequel TerminateIteration
      begin Inner; (* and prepare for termination *) end;
  loop
    declare (* inner-loop-block *)
      sequel UndoLastIteration
        begin Inner; (* and undo this iteration *) end;
    begin (* inner-loop-block *)
      (* take one iteration step *)
      :
      declare (* inner-block *)
        sequel Continue;
          begin ClosePrompter end;
        sequel Undo prefix UndoLastIteration;
          begin ClosePrompter end;
        sequel terminate prefix TerminateIteration;
          begin ClosePrompter; (* record last iteration *) end;
        sequel Abort prefix AbortSimulation;
          begin ClosePrompter end;
      begin (* inner-block *)
        DisplayPrompter('Select continuation:',
                        'Continue', Continue,
                        'Undo last iteration', Undo,
                        'Terminate', Terminate,
                        'Abort', Abort);
      end; (* inner-block *)
      (* prepare for next iteration *)
      :
    end; (* inner-loop-block *)
  end (* loop-block *)
  (* output simulation results and close simulation window *)
  :
end; (* simulation-block *)
```

Eksempel 4: Simuleringseksempel

*(*record last iteration*)*, og endelig *(*and prepare for termination*)*. Herefter vil simuleringen afsluttes med udskrivning af resultater og lukning af simuleringsvinduet, dvs. udførelse af *(*output simulation results and close simulation window*)*.

Abort er prefixet med *AbortSimulation*, der er erklæret i den yderste blok *simulation-block*. *AbortSimulation* er ikke prefixet med andre sequels og *Abort* vil derfor terminere *simulation-block* efter at have udført først *ClosePrompter*, og derefter *(*and close simulation window*)*. Simuleringen vil derfor termineres, uden at simuleringsresultaterne udskrives.

Dette eksempel viser ikke den fulde udtryksstyrke af forslaget. For en mere grundig diskussion henvises til reference /3/.

Statisk undtagelseshåndtering i objekt-orienteret programmering

Den naturlige fortsættelse af ovennævnte arbejder er at diskutere statisk undtagelseshåndtering i et sprog med kraftig understøttelse af objekt-orienteret programmering. Kravet til et sådant sprog må være, at det er et sprog med blok-struktur samt statisk navnebinding, idet sequel-begrebet benytter sig af disse mekanismer. Der er ikke ret mange sprog med kraftig understøttelse af objekt-orienteret programmering, som samtidigt er blok-strukturerede med statisk navnebinding. Af sprog kan nævnes Simula67, Beta og Galileo. Valget her faldt på Beta af flere grunde. For det første kan Beta ses som en udvidelse/generalisering af ideerne i Simula67. For det andet er Galileo rettet mod objekt-orienteret programmering udfra et funktionelt synspunkt, hvilket gør, at kontrol-begrebet er underspillet. Endelig er Beta udviklet i tilknytning til DAIMI i et fællesprojekt med deltagere fra DAIMI, AUC, Oslo Universitet, og Norsk Regnesentral. Fra DAIMI deltager lektor Ole Lehrmann Madsen. Dette giver en ekstra adgang til at diskutere detaljer i bla. implementationen udfra viden om den proto-type implementation, der foretages for øjeblikket.

Artiklen »*Static Exception Handling in Beta*« (ref./4/) består af fire afsnit. Det første afsnit diskuterer et ændringsforslag til Beta, der gør det muligt at benytte virtuel binding i indre blokke i handlingsdelen af en deskriptor. Det viser sig, at dette lader sig gøre, hvis man løsner en restriktion i Beta. Ændringen vil dog betyde et ekstra overhead under programudførelsen, men dette overhead pådrages kun de dele af programmet, der benytter faciliteten.

Herefter diskuteres indføringen af sequel-begrebet i den strengt sekventielle del af Beta. Programmeringssproget Beta er designet udfra en filosofi om et fåtal af meget generelle udtryksmidler, som kan anvendes ortogonalt, hvorved opnåes, at en lang række gængse udtryksmidler fra andre sprog lader sig udtrykke vha. disse få udtryksmidler. Det ville derfor være i modstrid med filosofien i Beta at indføre et nyt udtryksmiddel (sequel-begrebet) med egne regler. Forslaget går derfor ud på at designe et specielt pattern,[1] som så vidt muligt lader sig beskrive/implementere i Beta. Dette pattern skal så indgå på lige fod med øvrige patterns i Beta (altså følge de samme regler), hvorved ortogonaliteten bibeholdes. Afsnit 2.1 diskuterer betydningen og anvendelsen af et sådant pattern i Beta-programmer. Det viser sig, at sequel-begrebet lader sig introducere i Beta på denne måde uden at give anledning til større problemer. Herudover illustrerer afsnittet betydningen af at aktivere en sequel,

[1]Et pattern i Beta er en navngiven deskriptor, som kan benyttes til instantiering af objekter (instanser). Vha. pattern-begrebet kan man udtrykke, hvad der svarer til typer, procedurer og funktioner i andre programmeringssprog.

der er en attribut af en statisk instans (et problem, der ligeledes berøres kort i reference /2/, afsnit 2.7). Problemet har en klar betydning indenfor objekt-orienteret programmering, idet objekter her har selvstændig eksistens og selvstændig handling, og dermed behov for selvstændig undtagelseshåndtering. Den løsning, der bliver diskuteret i relation til Beta bærer præg af grundlæggende egenskaber ved Beta-sproget (mht. stak-organiseringen er der forskel på, hvorvidt en sequel kaldes fra objektets egen handlingsdel eller ved »remote access«). Problemet har således ikke kunnet finde sin ultimative løsning i Beta pgra. den valgte realisering af sequel-begrebet. Det springende punkt er, at prefix-hierarkiet for en sequel ikke automatisk giver anledning til at terminere de mellemliggende lag (de lag, hvori hvert led i prefix-hierarkiet er erklæret). Dette er en konsekvens af, at sequel-begrebet indføres som et predefineret pattern, der skal indgå på lige fod med øvrige patterns. En sådan automatisk terminering af de mellemliggende lag lader sig naturligvis implementere, men dette har ønsket om simpelhed i semantik afholdt mig fra. Det er således i visse tilfælde op til programmøren at terminere de mellemliggende lag, hvis det ønskes.

Afsnit 2.2 omhandler statisk undtagelseshåndtering i den multi-sekventielle del af Beta. Her diskuteres en række problemer, som ikke har været diskueret i det øvrige materiale. Problemstillingen er her, at når en undtagelse opstår, kan der være mere end ét kontrol-flow involveret. De eksisterende forslag til sproglige udtryksmidler til undtagelseshåndtering (eksempelvis i programmeringssproget Ada) tager det udgangspunkt, at en undtagelse, der forekommer, når flere kontrol-flow er involveret, er en undtagelse for samtlige disse kontrol-flow (mao. undtagelsen skal propageres ud langs samtlige disse kontrol-flow). I statisk undtagelseshåndtering hører en undtagelse til ét og kun ét kontrol-flow, og det er således op til programmøren at afgøre, hvad konsekvensen skal være for evt. andre kontrol-flow, der måtte være involveret, når undtagelsen forekommer. Dette giver en øget flexibilitet til at programmere andre undtagelseshåndteringsdiscipliner ved multiple kontrol-flow end de undtagelseshåndteringsdiscipliner, der er forudbestemte i eksempelvis Ada. Omvendt kan der opstå deadlock-situationer, men disse deadlock-situationer ville også kunne opstå i Beta-programmer, der ikke benytter sequel-begrebet som beskrevet her.

Afsnit 3 omhandler en mulig implementation af sequel-begrebet i Beta. Mao. diskuteres et forslag til, hvorledes det foreslåede sequel-pattern kan realiseres i basic Beta. Her argumenteres for, at det predefinerede pattern Sequel ikke umiddelbart lader sig implementere i basic Beta, men med mindre ændringer i basic Beta gives et konkret forslag til realisering.

Opsummering

Licentiatstudiet har været præget af to parallelle angrebsvinkler på programmeringssprog, nemlig en analyserende og en konstruerende angrebsvinkel. Resultatet af den analyserende angrebsvinkel er beskrevet i artiklen: *»A Conceptual Framework for Programming Languages«* (ref./1/), som er udarbejdet i fællesskab med Kristine Stougård Thomsen. Artiklen er en analyse af programmeringsprocessen mhp. at afklare programmeringssproglig understøttelse af programmeringsprocessen, og beskriver en sproguafhængig begrebsramme for diskussion af programmeringssprog og programmeringssproglige udtryksmidler. Analysen og begrebsrammen lægger stor vægt på programmeringssproglig understøttelse af abstraktion.

Det begrebsapparat, der opstilles i artiklen, er ikke fuldstændigt i den forstand, at man vha. dette begrebsapparat kan beskrive samtlige aspekter af et specifikt programmeringssprog. Begrebsapparatet skal betragtes som et supplement til andre begrebsapparater (eksempelvis semantiske og syntaktiske begrebsapparater), og anvendes parallelt med disse til at analysere specielt programmeringssprogets modelleringsstyrke med særlig vægt på

understøttelse af abstraktion. Vi betragter begrebsapparatet som værende et afbalanceret hele, som afspejler den nuværende indsigt i programmeringssproglig understøttelse af specielt abstaktion. Begrebsapparatet er som sådant stabilt, men det må naturligvis tages op til kritisk revision, såsnart en evt. ny indsigt omkring programmeringsprocessen opnåes.

Den konstruerende angrebsvinkel har fokuseret på design af et sprogligt udtryksmiddel til statisk undtagelseshåndtering. Resultatet af denne angrebsvinkel er beskrevet i artiklerne: »*Exception Handling — A Static Approach*« (ref./2/), »*A Hierarchical, Co-operative Exception Handling Mechanism*« (ref./3/), samt »*Static Exception Handling in Beta*« (ref./4/). Udgangspunktet for dette arbejde har været en analyse af eksisterende forslag til sproglige udtryksmidler til undtagelseshåndtering, der alle benytter en dynamisk bindingsregel. Analysen viste, at disse udtryksmidler, pga. denne dynamiske bindingsregel, ikke var integrerede i det øvrige programmeringssprog, hvori de tænktes anvendt. Det blev derfor interessant at designe et sproglige udtryksmiddel til undtagelseshåndtering, der benytter en statisk bindingsregel. Licentiatstudiet har vist, at dette har været muligt, og at det ligeledes er muligt at overføre statisk undtagelseshåndtering til objekt-orienteret programmering, eksemplificeret ved programmeringssproget Beta.

Licentiatstudiet har vist, at der findes et alternativ til de eksisterende dynamiske udtryksmidler til undtagelseshåndtering. Licentiatstudiet har ligeledes vist, at for de dynamiske forslag er undtagelseshåndtering ved at være et velafklaret begreb. Statisk undtagelseshåndtering er endnu ikke afprøvet under realistiske forhold (under design og programmering af større systemer). Der er dog ikke noget, der tyder på, at en sådan afprøvning ville støde på vanskeligheder. Det er blot vigtigt at være opmærksom på, at der skal anvendes et andet designkoncept for statisk undtagelseshåndtering end for dynamisk undtagelseshåndtering.

Indføring af statisk undtagelseshåndtering som beskrevet i reference /2/ i eksempelvis programmeringssproget Ada vil ikke volde større vanskeligheder. Indføring af statisk undtagelseshåndtering som beskrevet i reference /3/ i Ada vil heller ikke volde større vanskeligheder. Her bør man blot være opmærksom på, at man indfører sproglige udtryksmidler rettet mod objekt-orienteret programmering i et programmeringssprog, der primært er rettet mod algoritmisk programmering. Da objekt-orienteret programmering og algoritmisk programmering har mange fællestræk, kan samme programmeringssprog være rettet mod begge programmeringsstile (som eksempelvis Beta). Der vil derfor ikke være programmeringsstilistiske argumenter imod indføring af statisk undtagelseshåndtering som beskrevet i reference /3/ i Ada.

Indføring af statisk undtagelseshåndtering i Beta viste sig at være muligt ved simple modifikationer til basic Beta og ved tilpasning af sequel-begrebet til Beta's meget generelle struktur. Beta's generalitet viste sig at have både fordele og ulemper. Sequel-begrebet i Beta som beskrevet i reference /4/ kan betragtes som et endeligt forslag, hvor enkelte problemstillinger må afvente afklaring i forbindelse med konkret implementation og evt. justering i forbindelse med realistiske proto-type afprøvninger.

Referencer:

/1/ Jørgen Lindskov Knudsen, Kristine Stougård Thomsen,
A Conceptual Framework for Programming Languages,
DAIMI PB-192, Datalogisk afdeling, Aarhus universitet, april 1985.

/2/ Jørgen Lindskov Knudsen,
Exception Handling — A Static Approach,
Software — Practice & Experience, 14(5), 429-449, (maj 1984).

/3/ Jørgen Lindskov Knudsen,
A Hierarchical, Co-operative Exception Handling Mechanism,
DAIMI PB-204, Datalogisk afdeling, Aarhus universitet, januar 1986.

/4/ Jørgen Lindskov Knudsen,
Static Exception Handling in Beta,
DAIMI PB-214, Datalogisk afdeling, Aarhus universitet, juni 1986.