

Ada

En orientering

27. februar 1984

Forfattere: Jesper Jørgensen
Rolf Molich
Ole Oest

Dokument nr. DDC 02/1984-02-27

Copyright 1981, 1982, 1984 Dansk Datamatik Center

Indholdsfortegnelse

Indholdsfortegnelse

	Side
1. Indledning	1-1
2. DoD Ada Projektet	2-1
2.1 Kronologisk oversigt	2-2
2.2 Officielle Ada dokumenter	2-7
2.3 Omgivelsessystemet	2-8
2.4 Igangværende projekter	2-12
2.4.1 Godkendelse af Ada oversættere	2-12
2.4.2 Ada oversætter til PDP-11 og VAX ...	2-13
2.4.3 Komplet omgivelsessystem til IBM 370	2-13
2.4.4 Europæisk Ada oversætter	2-13
2.4.5 Europæisk Ada oversætter og omgivelsessystem	2-14
2.4.6 Andre vigtige projekter	2-14
2.5 Litteratur om DoD Ada projektet	2-15
3. DDC's Ada aktiviteter	3-1
3.1 Det dansk/italienske Ada projekt	3-2
3.1.1 Projektets indhold og omfang	3-3
3.2 DDC's Ada oversætter	3-5
3.2.1 Den resulterende oversætter	3-5
3.2.2 Boot-strap processen	3-7
3.2.3 Systematisk oversætterkonstruktion .	3-8
3.3 Litteratur	3-10
4. Introduktion til Ada	4-1
4.1 Simpelt eksempel	4-2
4.2 Datastrukturer	4-3
4.3 Kontrolstrukturer	4-9
4.4 Programstruktur	4-13
4.5 Afsluttende eksempel	4-15
4.6 Afledte egenskaber	4-16
4.7 Sammenligning mellem Ada og andre programmeringssprog	4-18
5. Parallel programmering i Ada	5-1
5.1 Kommunikation og synkronisering	5-3
5.2 Tidsforsinkelser	5-9
6. Modulær programmering i Ada	6-1
6.1 Programmoduler	6-2
6.2 Separat oversættelse	6-7
6.3 Generiske moduler	6-9

Indholdsfortegnelse

7.	Grænseflader til omgivelser	7-1
7.1	Inddata/uddata	7-1
7.2	SEQUENTIAL_IO og DIRECT_IO pakkerne	7-3
7.3	TEXT_IO pakken	7-6
7.4	LOW_LEVEL_IO pakken	7-7
7.5	Grænseflader til andre programmeringssprog	7-8
7.6	Grænseflader til den underliggende maskine	7-9
	7.6.1 Repræsentationsspecifikationer	7-9
	7.6.2 Afbrydelser	7-9
8.	Ada lærebøger	8-1

Indledning

1. Indledning

Dette dokument er udarbejdet af Dansk Datamatik Center til et heldagsseminar om det nye programmeringssprog Ada. Seminaret formål er at orientere om Ada, sprogets historie, dets karakteristiske egenskaber og dets anvendelsesmuligheder, således at deltagerne i seminaret kan danne sig et indtryk af, hvorvidt Ada er relevant for de områder, de arbejder med.

Til hvert af seminarets foredrag er der udarbejdet et kapitel, som giver en afrundet beskrivelse af foredragets emne. De enkelte kapitler kan læses hvert for sig.

Der indledes med en kort orientering om tilblivelseshistorien for sproget Ada. Herefter gennemgås et dansk/italiensk projekt, som omfatter udvikling af en Ada oversætter og et Ada omgivelsessystem. Dansk Datamatik Center og Christian Rovsing A/S er med i dette projekt. Kapitel 4 indeholder en introduktion til Ada, hvor hovedvægten er lagt på de nye faciliteter i sproget. De næste 3 kapitler omtaler mere detaljeret de 3 væsentligste ny-skabelser i Ada: faciliteter til parallel programmering, modular programmering og i/o systemet. Det sidste kapitel giver en oversigt over eksisterende lærebøger og introducerende tekster om Ada.

Kapitel 3 og 6 er skrevet af Ole Oest. Kapitel 5 og 7 er skrevet af Jesper Jørgensen (Christian Rovsing A/S), mens kapitel 2, 4 og 8 er skrevet af Rolf Molich, der også har redigeret dokumentet.

DoD Ada projektet

As long as there were no machines, programming was no problem at all; when we had a few weak machines, programming became a mild problem, and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them - it has created the problem of using its products.

E.W.Dijkstra
Turing Award Lecture 1972

2. DoD Ada Projektet

Programmeringssproget Ada er udviklet på initiativ af det amerikanske forsvarsministerium (Department of Defense, DoD). Ada skal være det fremtidige fælles programmeringssprog for alle programmeringsopgaver for den amerikanske hær, flåde og luftvåben.

Baggrunden for bestræbelserne for at udvikle et fælles, standardiseret programmeringssprog er et ønske fra DoD's side om at reducere sine programmeludgifter, som i 1973 androg 3 - 3½ milliard dollars. En af årsagerne til de store programmeludgifter er anvendelsen af over 400 sprog og dialekter på mere end 200 forskellige typer af datamater, der alle er mere eller mindre inkompatible. Det store antal forskellige sprog og dialekter medfører at de forsøg der gøres på at udvikle tidssvarende værktøjer til programmeludvikling og vedligeholdelse kun er til gavn for en begrænset kreds af brugere.

Det er DoD's opfattelse at et fællessprog for programmeludvikling inden for det amerikanske forsvar vil

- nedsætte uddannelsesomkostninger,
- muliggøre genbrug af programmer eller programmoduler,
- muliggøre centraliseret programmeludvikling og fælles programbiblioteker,
- gøre det økonomisk attraktivt at konstruere et effektivt udviklings- og vedligeholdelsesmiljø.

Hjælpeprogrammel, som f.eks. oversættere til specialiserede programmeringssprog, kan udvikles i fællessproget, hvilket vil gøre ethvert hjælpeværktøj tilgængelig på enhver datamat efter at nogle få hjælpeprogrammer (kodegenerator/simulator) er tilpasset den nye datamat.

DoD Ada projektet

2.1 Kronologisk oversigt

- 1974 En række arbejdsgrupper inden for det amerikanske forsvar foreslår at et fælles programmeringssprog standardiseres til fremtidig udvikling af større programmel-projekter i forsvaret.
- 1975, januar Viceforsvarsministeren beslutter at DoD ikke må bevillige midler til implementering af nye programmeringssprog i større forsvarsprojekter før fællessprogsproblematikken er afklaret.
- 1975, april Strawman specifikationer, der foreslår en række krav til det nye sprog, udsendes.
- 1975, august Woodenman specifikationer udsendes. Kravene til det nye sprog udvides og præciseres. Denne specifikation er den sidste der opstiller nye generelle krav til sproget.
- 1976, januar Tinman specifikationer udsendes. En lang række urealistiske og ubegrundede krav til det nye sprog fjernes.
- 1976 En lang række eksisterende sprog vurderes mod Tinman specifikationer. Der er offentliggjort vurderinger af Fortran, Cobol, PL/1, HAL/S, Tacpol, CMS-2, CS-4, SPL-1, J3B, Algol 60, Algol 68, Coral 66, Pascal, Simula 67, Lis, LTR, RTL/2, Euclid, PDL2, Pearl, Moral og EL-1.
- Vurderingerne gav følgende konklusioner:
1. Intet sprog er tilfredsstillende som det er.
 2. Målet er ét fællessprog.
 3. Forskningen er så vidt fremskreden at kravene kan opfyldes.
 4. Udviklingen bør ske på basis af et passende, eksisterende sprog.
- 1976, nov. De militære værn under DoD enes om en foreløbig liste over godkendte programmeringssprog. Disse sprog er de eneste der fremover må anvendes til nye opgaver. Listen omfatter CMS-2, SPL-1, Tacpol, J3 Jovial, J73 Jovial, ANSI Cobol og ANSI Fortran.

DoD Ada projektet

1977, januar Ironman specifikationer udsendes. Disse specifikationer er ganske korte og er organiseret som en håndbog over det nye sprog, der får tildelt arbejdsbetegnelsen DoD-1. Beskrivelsen fastlægger strukturen af det nye sprog uden at diktere detaljer vedrørende designet.

1977, juli Revision af Ironman specifikationer.

1977, august Sytten firmaers tilbud på et indledende design af det nye sprog vurderes af DoD. Fire firmaer udvælges til at udvikle hver sit design, nemlig

- CII-Honeywell-Bull (Grøn)
- Intermetrics (Rød)
- SRI-International (Gul)
- SofTech (Blå)

Farverne er de pseudonymer som hvert design blev udviklet under for at sikre en upartisk vurdering af de forskellige forslag.

1978, april To designforslag, det grønne og det røde, udvælges som de bedste. Firmaerne bag disse forslag får bevillinger til at færdiggøre deres design og udvikle en testoversætter svarende til deres designforslag.

1978, juni Steelman specifikationer udsendes. Disse specifikationer giver faste men dog ikke autoritære retningslinier for den endelige udarbejdelse af sproget.

1979, januar Pebbleman specifikationer, som skitserer kravene til omgivelsessystemet, udsendes.

1979, marts De endelige designforslag afleveres.

1979, maj Efter omfattende vurderinger foretaget af mere end 50 forskellige hold af eksperter, udråbes det grønne design til vinder. Det grønne design er udarbejdet af et designhold hos CII-Honeywell-Bull i Paris under ledelse af Jean Ichbiah.

Det grønne sprog omdøbes til Ada, til ære for lady Ada Augusta, Countess of Lovelace, som var verdens første programmør, idet hun programmerede Charles Babbages første primitive regnemaskiner. En udmærket beskrivelse af den historiske person lady Ada Augusta findes i (Huskey 80), se afsnit 2.5.

DoD Ada projektet

Bemærk at Ada er et navn - ikke en forkortelse. Derfor skal navnet Ada altid skrives med store og små bogstaver.

- 1979, juni ACM Sigplan Notices publicerer en beskrivelse af Ada sproget i to bind, en referencenhåndbog og en begrundelse for de enkelte sprogkonstruktioner. Ada sproget bliver herved kendt i vide kredse.
- 1979, sept. SofTech påbegynder udviklingen af Ada Compiler Validation Capability (se afsnit 2.4).
- 1979, sept. EF beslutter at skrinlægge sine bestræbelser for udvikling af et nyt fælleseuropæisk programmeringssprog (ESL), og vælger i stedet at give Ada sproget sin fulde støtte. ESL projektet har på dette tidspunkt kørt i 4-6 år uden særlig succes.
- 1979, oktober Over 100 programmørgrupper på universiteter, industrielle foretagender og i det offentlige rapporterer deres erfaringer med programmering af et 1000 - 5000 liniers program i Ada på en konference i Boston. Erfaringerne er som helhed positive.
- 1980, februar Stoneman specifikationerne udsendes. Disse specifikationer indeholder en række overordnede krav til omgivelsessystemet (se afsnit 2.3).
- 1980, marts SofTech påbegynder udvikling af en Ada oversætter til VAX 11, PDP 11/70 (begge fra Digital Equipment Corporation), og GYK 12 datamaterne efter at have modtaget en kontrakt fra US Army.
- 1980, juli Den første fuldstændige referencemanual for Ada sproget udsendes. Ind til sommeren 1981 troede man, at denne udgave af referencemanualen var den endelige udgave, og at udviklingen af Ada sproget hermed var afsluttet.
- 1980, august Der oprettes en central database med spørgsmål og svar vedrørende uklarheder i Ada referencemanualen. I de følgende 14 måneder indløber over 1.000 spørgsmål til dette system.

DoD Ada projektet

- 1980, sept. INRIA, der er underleverandør til CII-Honeywell-Bull, afleverer et udkast til en formel beskrivelse af Ada sproget til DoD. Beskrivelsen omfatter ikke tasking.
- 1980, oktober Professor C.A.R. Hoare, Oxford, angriber i sin vittige og sarkastiske Turing Award Lecture (Hoare 81) Ada sproget, bl.a. for at være uoverskueligt og umuligt at implementere korrekt.
- 1980, dec. To europæiske konsortier, bestående af hhv. Alsys/CII-Honeywell-Bull/Siemens og DDC/Christian Rovsing/Olivetti, får af EF-kommissionen bevilling til udvikling af hver sin europæiske Ada oversætter.
- 1981, april Ada sprogbeskrivelsen fremsendes til ANSI (American National Standards Institute) med henblik på standardisering.
- 1981, oktober ANSI afviser at standardisere Ada, bl.a. med den begrundelse at den eksisterende referencemanual er for upræcis. Selve sproget har man dog ingen væsentlige indvendinger imod.
- 1981, dec. DoD lader Ada registrere som beskyttet varemærke. Om konsekvenserne af dette se afsnit 2.4.1, bemærkningerne til SofTech's Ada Compiler Validation Capability.
- 1982, marts US Air Force tegner kontrakt med Intermetrics om konstruktion af et komplet omgivelsessystem inklusive Ada oversætter (se afsnit 2.4.5).
- 1982, juli En revideret referencemanual for Ada sproget udsendes. I forhold til referencemanualen fra juli 1980 er teksten totalt revideret, mens selve Ada sproget kun har undergået mindre ændringer.
- Den reviderede referencemanual forelægges ANSI med henblik på standardisering.
- 1983, januar ANSI godkender den reviderede referencemanual for Ada som amerikansk standard (ANSI/MIL-STD-1815A).
- For at skabe arbejdsro for de firmaer som konstruerer Ada oversættere lover DoD, at Ada tidligst vil blive revideret i 1988.

DoD Ada projektet

- 1983, februar New York Universitys Ada oversætter godkendes (valideres). Denne oversætter er den første der bliver godkendt som fuld Ada oversætter, men den er for ineffektiv til at få nogen kommerciel betydning.
- 1983, juni ROLM Corporation validerer sin Ada oversætter. Denne oversætter valideres kort tid senere også på den funktionelt identiske MV8000 datamat ("Eagle") fra Data General.
- 1983, august Western Digital validerer sin Ada oversætter. Denne oversætter bliver godkendt på trods af, at den ikke kan oversætte programmer som anvender Adas i/o pakke.

DoD Ada projektet

2.2 Officielle Ada dokumenter

Nedenstående oversigt opsummerer de vigtigste officielle dokumenter som er udsendt i Ada projektet. Dokumenterne i parentes er annonceret, men endnu ikke udsendt.

<u>År</u>	<u>Ada sproget</u>	<u>Omgivelsessystemet</u>	<u>Metoder</u>
1975	Strawman Woodenman		
1976	Tinman		
1977	Ironman	Sandman	
1978	Steelman		
1979	Ada Rationale I	Pebbleman	
1980	Referencemanual I Formel beskrivelse Implementers' Guide I	Stoneman	
1981			
1982	Referencemanual II		Methodman
1983	Referencemanual III (ANSI standard)		
1984	(Ada Rationale II) (Implementers' Guide II)		

DoD Ada projektet

2.3 Omgivelsessystemet

På det nuværende tidspunkt, hvor udviklingen af Ada sproget er afsluttet, er man begyndt at interessere sig for udviklingen af et integreret miljø for udvikling og vedligeholdelse af Ada programmer.

Udarbejdelsen af specifikationer for omgivelsessystemet ("Environment System") er foregået efter samme princip som udviklingen af Ada sproget, dvs. som en konkurrence mellem en række kommercielle firmaer.

Kravspecifikationerne til omgivelsessystemet findes i Stoneman rapporten, som blev udsendt af DoD i februar 1980.

For at kunne forstå filosofien bag Stoneman rapporten er det nødvendigt at huske, at DoD's primære interesse er at udvikle et system, der er velegnet til udvikling af militære programsystemer. Militære programsystemer udføres ofte på datamater ("måldatamater"), der er forskellige fra dem de blev udviklet og evt. afprøvet på ("værtssdatamater"). Dette skyldes bl.a. at en stor del af militærets datasystemer er primitive, indlejrede datamater ("Embedded Computer Systems"), dvs. systemer der kun er en lille del af et større system (hangarskib, raket, radarsystem osv.). Indlejrede datamatsystemer har ofte kun rudimentære, specialudviklede styresystemer, der skal udvikles som en del af projektet.

Standardiseringen af sprog og omgivelser vil medføre et større kommercielt udbud af hjælpeværktøjer, der vil kunne anvendes på alle datamater, hvor det standardiserede system er implementeret.

DoD Ada projektet

Stoneman rapporten forslår en systemstruktur som kan karakteriseres ved følgende 4 niveauer:

Niveau 0: datamat, grundlæggende styresystem, og filsystem (databasesystem).

Niveau 1: Grundlæggende Ada snitflade ("Kernel Ada Programming Support Environment", KAPSE), som fastlægger en standardiseret, maskinuafhængig programmelsnitflade til systemet. Denne del af systemet er den eneste, der skal omprogrammeres når systemet flyttes til en anden datamat.

Stoneman kræver, at KAPSE snitfladen skal specificeres i Ada. Dette er naturligt, idet Adas modulfacilitet hermed kan anvendes til at specificere snitfladerne på en naturlig og simpel måde.

Alt programmel over KAPSE niveauet skal skrives i Ada. Programmel under KAPSE niveauet kan skrives i Ada, men kan også være skrevet i andre sprog.

KAPSE snitfladen muliggør simpel flytning af programmel over KAPSE niveauet mellem datamater af forskellige fabrikater.

Niveau 2: Minimum Ada Programmeludviklings Miljø ("Minimum Ada Programming Support Environment", MAPSE). Dette niveau omfatter det minimale sæt af hjælpeprogrammer som er nødvendigt til udvikling, udførelse og vedligeholdelse af Ada programmer. Alle MAPSE programværktøjer skal være skrevet i Ada.

Ethvert MAPSE system skal kunne vedligeholde og udvide sig selv. Dette betyder, at enhver værtsdatamat også er en måldatamat og at Ada oversætteren skal være skrevet i Ada. Stoneman specifikationerne tager ikke stilling til om andre programmeringssprog skal/må være implementerede.

De krævede hjælpeprogrammer er editor, Ada oversætter, debugger, linker/loader, kommandofortolker, versionskontrolprogram ("configuration control program"), udskriftsprogram ("prettyprinter") og diverse nytteprogrammer til filsystemet.

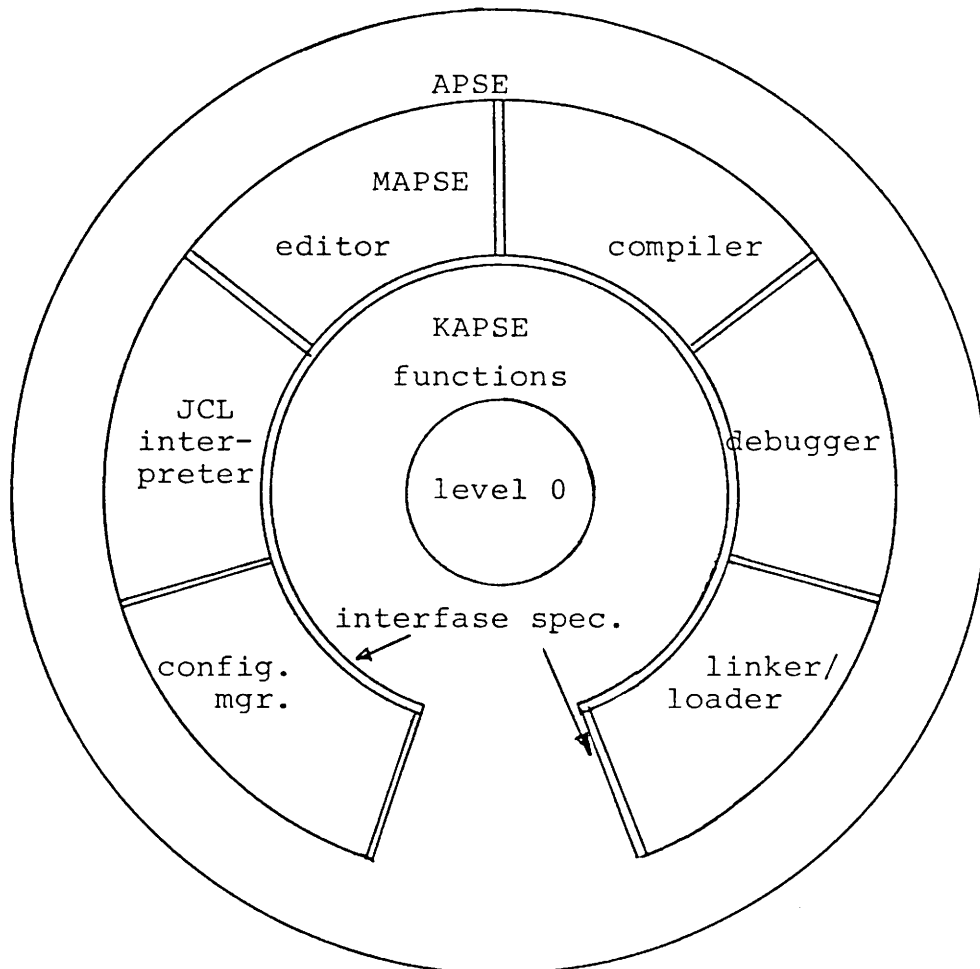
DoD Ada projektet

Hvis en planlagt standardisering af MAPSE programmernes brugersnitflade kan gennemføres, vil oplæringstiden blive reduceret væsentligt, når programmører overføres til arbejde ved en datamat af en type, som de ikke tidligere har arbejdet med, men som har en standard MAPSE snitflade.

Niveau 3: Ada Programmeludviklings Miljø ("Ada Programming Support Environment", APSE), som indeholder udvidelser af MAPSE værktøjssættet. På dette niveau findes værktøjer der f.eks. understøtter specielle anvendelser eller programudviklingsmetoder.

Eksempler på APSE programmel: Syntaksbaseret editor, dokumentationssystem, samt statiske og dynamiske måleværktøjer.

Strukturen for systemet fremgår af følgende diagram:



DoD Ada projektet

Databasen er et centralt begreb i et APSE system. Databasen anvendes til opbevaring af al information hørende til et projekt i hele projektets levetid.

Enhver selvstændig samling af informationsenheder i databasen kaldes et objekt. Hvert objekt kendes på sit entydige navn. Databasesystemet skal tillade relationer mellem objekter.

Alle objekter i databasen har entydige navne. Der kan imidlertid forekomme grupper af objekter som er forskellige versioner af samme grundobjekt. Inden for en sådan gruppe kan brugeren angive at et bestemt objekt er det foretrukne, som skal anvendes hvis brugeren ikke udtrykkelig anmoder om et specifikt objekt.

Det er muligt at opdele informationen i databasen i sektioner ("partitions"), f.eks. efter projekter. Det er muligt at begrænse adgangen til en sektion såvel som til et enkelt objekt. Sektionerne er den groveste inddeling af databasen.

Databasen tillader oprettelse og styring af konfigurative objekter, dvs. objekter der er samlinger af vilkårlige andre objekter i databasen. Konfigurative objekter kan optræde i forskellige versioner. Konfigurative objekter kan f.eks. beskrive forskellige udgaver af et styresystem.

De vigtigste oplysninger om et objekt som opbevares i databasen er afledningsoplysningerne. Disse oplysninger fortæller hvorledes objektet blev skabt, hvilken information der blev anvendt til at skabe objektet, samt hvorledes objektet anvendes. For et oversat program vil man f.eks. opbevare oplysninger om den tilgrundliggende programtekst, hvilken version af oversætteren der blev benyttet, hvilke optioner der blev valgt osv.

Databasen sikrer, at alle objekter der er nødvendige for at genskabe et bestemt objekt forbliver i databasen sålænge det pågældende objekt forefindes i databasen.

Databasen muliggør automatisk regenerering af et programsystem styret af det tilsvarende konfigurative objekt, hvis et af de objekter, der indgår i systemet, ændres. Hvis man således f.eks. modtager en fejlrapport vedrørende et modul, som indgår i en række programmer, kan systemet, når fejlen er rettet, automatisk oversætte og linke de berørte programmer.

DoD Ada projektet

2.4 Igangværende projekter

Der er iværksat en lang række Ada projekter. Den følgende oversigt omfatter nogle af de vigtigste.

2.4.1 Godkendelse af Ada oversættere

DARPA (Defense Advanced Research Projects Agency) har givet det amerikanske programfirmas SofTech en kontrakt på udvikling af en "Ada Compiler Validation Capability (ACVC)". ACVC er en omfattende samling af testprogrammer som med en rimelig grad af sandsynlighed kan godtgøre, om en forelagt oversætter arbejder i overensstemmelse med specifikationerne for Ada sproget.

Godkendelsesproceduren for en Ada oversætter består i, at leverandøren over for repræsentanter for et officielt testcenter demonstrerer, at oversætteren fejlfrit kan oversætte og udføre samtlige testprogrammer. Denne proces kaldes "validering".

Testprogrammerne kontrollerer blandt andet, at oversætteren er stand til at udpege alle fejl i et Ada program, at oversætteren ikke accepterer udvidelser af Ada sproget, samt at der ikke er foretaget vilkårlige indskrænkninger i sproget. Kun visse delmængder af Ada forventes at blive tilladt.

Kun oversættere, der har opnået en godkendelse, må betegnes Ada oversættere. Dette kan DoD styre, idet man har ladet betegnelsen Ada registrere som beskyttet varemærke både i USA og i en række europæiske lande.

En godkendelse er gyldig i to år, også hvis der i denne periode foretages ændringer i oversætteren. Herefter skal oversætteren påny underkastes godkendelsesproceduren med den nyeste version af testprogrammerne.

I øjeblikket består ACVC af mere end 3.000 programmer med tilsammen mere end 200.000 linier kildetekst. Alle testprogrammerne er principielt offentligt tilgængelige, ligesom enhver principielt kan foreslå nye testprogrammer.

Til vejledning for oversætterkonstruktører foreligger en "Implementers' Guide", som er en kritisk gennemgang af Ada sprogbeskrivelsen. "Implementers' Guide" forsøger at klarlægge de problemer, som vil kunne opstå under implementeringen af en Ada oversætter.

Der er fremskredne planer om et europæisk testcenter.

DoD Ada projektet

2.4.2 Ada oversætter til PDP-11 og VAX

US Army har givet SofTech en kontrakt på udvikling af en Ada oversætter til følgende måldatamater:

- VAX 11/780 med VAX/VMS OS
- VAX 11/780 uden styresystem
- PDP 11/70 med UNIX version 6
- GYK 12 uden styresystem.

(VAX og PDP er registrerede varemærker for Digital Equipment Corporation).

Udviklingen omfatter også et omgivelsessystem, Ada Language System (ALS). Udviklingen af dette system sker under hensyntagen til Stoneman specifikationerne.

SofTech forventede oprindeligt, at projektet ville blive afsluttet i foråret 1983. Projektet er imidlertid blevet voldsomt forsinket. I dag forventes systemet til VAX færdig i oktober 1984, mens systemet til de militære datamater forventes færdig i december 1985.

2.4.3 Komplet omgivelsessystem til IBM 370

US Air Force har i april 1982 skrevet kontrakt med det amerikanske programmefirma Intermetrics om udvikling af et komplet MAPSE system inklusive Ada oversætter. Systemet, der betegnes Ada Integrated Environment (AIE), skal let kunne flyttes til nye værtsdatamater eller omstilles til nye måldatamater. Systemet skal indledningsvis implementeres på IBM 370 under VM styresystemet.

Hele projektet forventes afsluttet i slutningen af 1984.

2.4.4 Europæisk Ada oversætter

EF kommissionen gav i december 1980 et konsortium bestående af Alsys/CII-Honeywell-Bull/Siemens bevilling til konstruktion af en Ada oversætter til store datamater. Firmaet Alsys (Ada language systems) ledes af Jean Ichbiah, som er "far" til Ada, idet han var projektleder på CII-Honeywell-Bulls designhold for Ada sproget.

Bevillingen udgør ca. 29 millioner Dkr. svarende til 50% af projektomkostningerne.

DoD Ada projektet

Oversætteren skal indledningsvis implementeres på CII-Honeywell-Bull Level 64 og Siemens 7760 datamaterne. Alsys vil også udvikle oversættere til Motorola 68000 og Intel 8086.

Projektet forventes afsluttet i midten af 1984.

2.4.5 Europæisk Ada oversætter og omgivelsessystem

EF kommissionen har i december 1980 givet et konsortium bestående af DDC/Christian Rovsing/Olivetti med det engelske firma SDL som underleverandør bevilling til konstruktion af en Ada oversætter til minidatamater samt et KAPSE/MAPSE system i overensstemmelse med Stoneman specifikationerne.

Bevillingen udgør ca. 21 millioner Dkr. svarende til 50% af projektomkostningerne.

Den indledende implementering vil ske på Christian Rovsing CR80D og Olivetti S6000 datamaterne.

Projektet forventes afsluttet i midten af 1984.

2.4.6 Andre vigtige projekter

Ud over de ovenfor nævnte projekter er der følgende vigtige projekter, som forventes at resultere i en eller flere godkendte Ada oversættere:

<u>Organisation</u>	<u>Værts- og måldatamat familie</u>
Telesoft	DEC VAX, IBM, Motorola 68000
Universität Karlsruhe	DEC VAX, IBM-370, Siemens 7000

DoD Ada projektet

2.5 Litteratur om DoD Ada projektet

- (Dijkstra 72) E.W. Dijkstra
"The Humble Programmer"
Turing Award Lecture
Communications of the ACM, Oktober 1972
- (DoD 80) Department of Defense
"Requirements for Ada Programming
Support Environments - Stoneman"
Department of Defense, Februar 1980
- (Fisher 78) David A. Fisher
"DoD's Common Programming Language
Effort"
Computer, Marts 1978, pp 24-33
- (Gilbert 80) Roger Gilbert, Vic Stenning
"Ada Environment Procurement in the USA"
European Camac Association Newsletter,
Vol.1, No.1, December 1980, pp 7-8
- (Goodenough 80) John B. Goodenough
"The Ada Compiler Validation Capability"
ACM Sigplan Notices,
Vol.15, No.11, November 1980, pp 1-8
- (Hoare 81) C.A.R. Hoare
"The Emperor's Old Clothes"
Turing Award Lecture
Communications of the ACM,
Vol. 24, no. 2, February 1981
- (Huskey 80) Velma R. Huskey, Harry D. Huskey
"Lady Lovelace and Charles Babbage"
Annals of the History of Computing
Vol.2, No.4, October 1980
- (Wegner 80) Peter Wegner
"The Ada Language and Environment"
ACM SIGSOFT Software Engineering Notes,
Vol.5, No.2, April 1980, pp 8-14

DDC's Ada aktiviteter

3. DDC's Ada aktiviteter

DDC er involveret i et antal Ada aktiviteter, der alle er baseret på resultaterne af DDC's del af det dansk/italienske Ada projekt, jfr. afsnit 3.1.

Den ene del af aktiviteterne vil resultere i Ada oversættere i følgende varianter:

<u>Variant</u>	<u>Værtsmaskine</u>	<u>Målmaskine</u>
1	Christian Rovsing CR80	CR80
2	Olivetti M40	M40
3	NOKIA MPS 10	MPS 10
4	VAX/VMS	CR80
5	VAX/VMS	VAX/VMS

Med værtsmaskine menes den maskine, hvorpå oversætteren kører, medens målmaskinen er den maskine, hvortil der genereres kode.

I variant 2 og 3 udføres en del af implementeringen af kodegeneratoren af Olivetti, Italien, henholdsvis NOKIA, Finland.

Herudover studeres mulighederne for at generere kode til visse mikrodatamater, bl.a. Intel's 8086 serie.

Den anden del af Ada aktiviterne har været et studie af, hvorvidt oversættere og værktøjer for CHILL og Ada kan forenes i samme programmeludviklingsmiljø, nærmere betegnet i den "MAPSE", der beskrives i afsnit 3.1.1. Som basis for dette arbejde har DDC benyttet de specifikationer, der stammer fra udviklingen af DDC's CHILL og Ada oversættere. Begge oversætterprojekter blev grebet an med metoden beskrevet i afsnit 3.2.3. Dette gjorde det muligt at foretage sammenligningen af CHILL og Ada på det abstrakte niveau, hvor begge sprog og oversættere er definerede, henholdsvis specificerede.

DDC's Ada aktiviteter

3.1 Det dansk/italienske Ada projekt

Det dansk/italienske Ada projekt, som udføres af Olivetti, Dansk Datamatik Center og Christian Rovsing A/S, resulterer i et komplet, flytbart Ada system specielt konstrueret med henblik på mindre datamater (16 bits) med begrænset adresserum.

Ada systemet består af et omgivelsessystem ("programmel-udviklingsmiljø") indeholdende oversætter og køretidssystem samt hjælpeprogrammer i overensstemmelse med Stoneman specifikationerne. Ved udviklingen af oversætteren og køretidssystemet benyttes formelle og systematiske specifikations- og implementeringsmetoder.

Projektet støttes af EF-kommissionen med ca. 21 millioner kroner, som udgør halvdelen af projektomkostningerne. Yderligere to større Ada projekter støttes under EF-kommissionens 4-års program (1979 - 1983), nemlig:

- fransk/tysk projekt udført af CII Honeywell Bull/Alsys/Siemens resulterende i oversætter og køretidssystem til større datamater
- engelsk/tysk studie udført af NPL/GMD/BNI omkring oprettelse af et europæisk godkendelsescenter for Ada oversættere.

Udveksling af ideer/erfaringer sker i "Ada-Europe", som er en arbejdsgruppe nedsat af EF-kommissionen med repræsentanter for de fleste vesteuropæiske Ada projekter. Desuden deltager DDC i Ada konferencer i Europa og USA.

DDC's Ada aktiviteter

3.1.1 Projektets indhold og omfang

De tre samarbejdspartnere udvikler et "minimum Ada programudviklingsmiljø", kaldet MAPSE i Stoneman terminologi, jfr. forrige kapitel om DoD Ada projektet. Projektets hovedbestanddele er følgende:

- 1) Specifikation og implementering af en standard grænseflade til den underliggende datamats operativsystem og filsystem (KAPSE), herunder specifikation af KAPSE i form af en Ada pakke. I KAPSE indgår en database som en meget væsentlig komponent.
- 2) Specifikation og implementering i Ada af et minimalt sæt programmeludviklingsværktøjer:
 - tekst editor
 - debugger
 - database håndtering
 - kommandofortolker
 - formatteringsprogram
 - lænkeprogram
 - håndtering af modulbibliotek
- 3) Ada oversætter skrevet i Ada bestående af:
 - "Front End", der oversætter Ada til en mellemform
 - "Back End", der genererer såkaldt A-kode fra mellemformen til Christian Rovsing's CR80 og maskinkode til Olivetti's M40 datamat
 - Håndtering af separat oversættelse
- 4) Køretidssystem inklusive en virtuel maskine:
 - Design af flytbart køretidssystem (A-kode maskine, monitor til at håndtere parallelisme, I/O system)
 - Implementering af køretidssystem på CR80D og Olivetti M40 datamaterne

DDC's Ada aktiviteter

- 5) Studie af distribuerede systemers krav til og indflydelse på KAPSE og MAPSE

En vigtig del af projektets punkt 3 og 4 er en demonstration af formelle, systematiske metoders slagkraft i forbindelse med udvikling af oversættere.

Punkt 1, 2 og 5 udføres af Olivetti, punkt 3 af DDC og punkt 4 af Christian Røvsing A/S.

Projektets samlede størrelse er ca. 1000 personmåneder, og projektperioden er 3 år, nemlig 1981 til 1984.

DDC's Ada aktiviteter

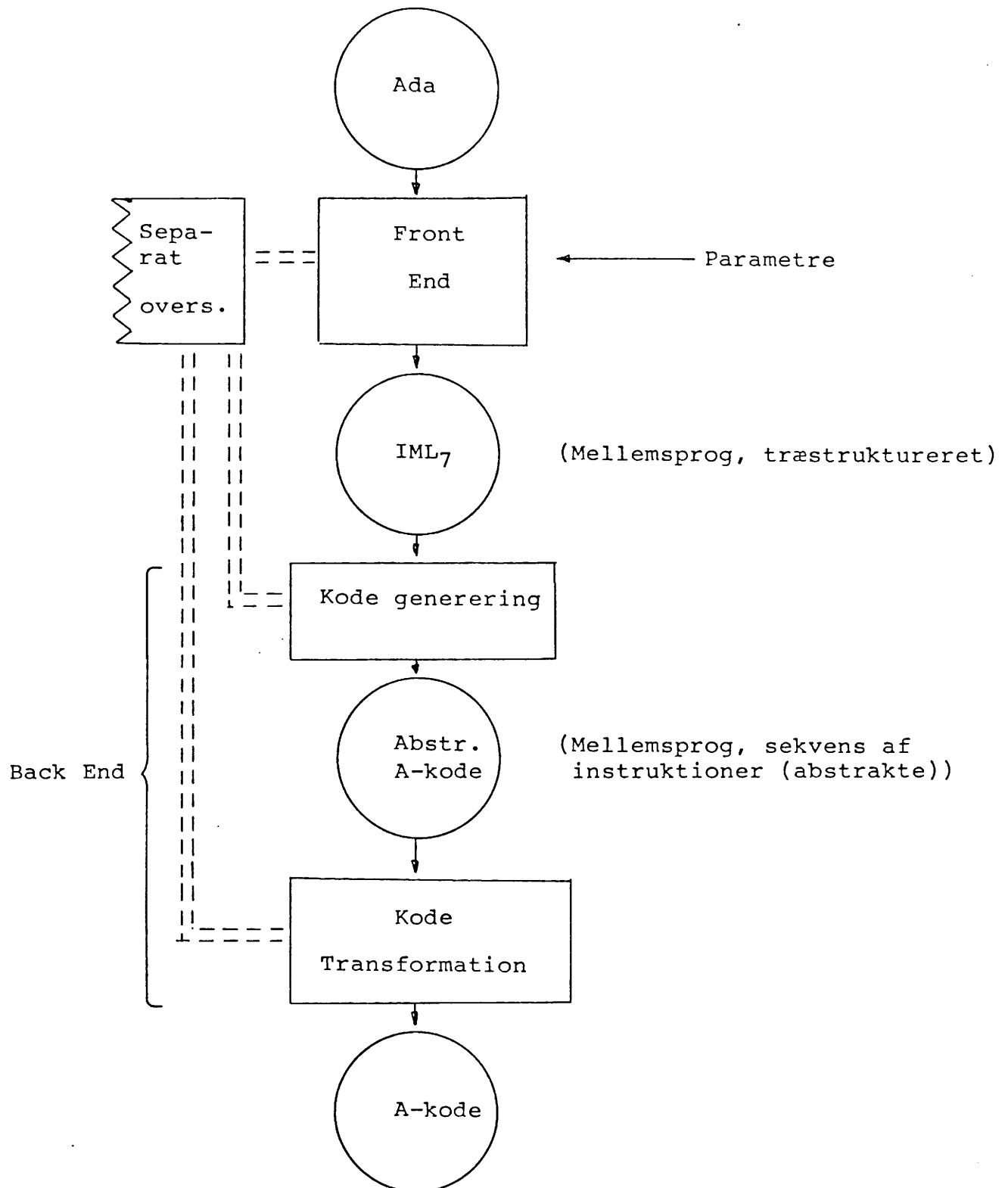
3.2 DDC's Ada oversætter3.2.1 Den resulterende oversætter

Der kan identificeres 3 hovedkomponenter i oversætteren, nemlig:

- en "Front End", der oversætter Ada programmer til en mellemform og som desuden udfører den kontrol af Ada programmets korrekthed, som kan ske under oversættelsen. Ada er designet på en sådan måde, at en stor del af denne kontrol kan foretages på oversættelsestidspunktet snarere end på kørselstidspunktet
- en "Back End" bestående af to passager, der via "abstrakt A-kode" genererer A-kode ud fra mellemformen
- en komponent, der håndterer separat oversættelse.

Muligheden for separat oversættelse er defineret som en del af sproget, således at der kan foretages fuld kontrol også af grænsefladerne mellem separat oversatte moduler.

DDC's Ada aktiviteter



DDC's Ada aktiviteter

Et væsentligt krav til oversætteren er, at den skal være flytbar. Dette opnås ved at lade oversætteren generere en virtuel kode, såkaldt A-kode, der svarer til Pascals P-kode. A-kode tillader også udførelse af processer ("tasks") i parallel. Oversætteren og dermed også MAPSE værktøjerne kan da flyttes til en ny maskine ved at implementere en A-kode fortolker med køretidssystem samt det standardiserede KAPSE på den nye maskine. Skulle man senere ønske at generere maskinkode, kan en ny Back End hertil udvikles i Ada ved hjælp af det A-kode baserede system. Maskinkode kan genereres ud fra abstrakt A-kode (benyttet til VAX) eller IML7 (benyttet til MPS 10).

A-koden giver også andre fordele end god flytbarhed: A-kode er mere kompakt end traditionel maskinkode, da A-kode instruktionerne er mere slagkraftige - designet af programmell- snarere end materiel-specialister, og specielt tilpasset Ada. Det vil være muligt at implementere A-kode maskiner direkte ved hjælp af mikro kode - herved opnås en maskine, der er specielt velegnet til Ada. A-kode tillader en enkel og billig implementering af en slagkraftig debugger.

Det var en del af det samlede projekt at definere A-koden under passende hensyntagen til Ada, til de aktuelle målmaskiner og til andre A-koder, der måtte blive defineret. A-koden blev defineret af Christian Rovsing A/S (CR 83).

3.2.2 Boot-strap processen

Oversætteren er skrevet i en delmængde af Ada og udvikledes på CR80D datamaten. En af de første aktiviteter i projektet var konstruktion af et program, der transformerer programmer og moduler skrevet i denne delmængde af Ada til sproget SWELL, der kan oversættes på CR80D. Dette program er senere blevet flyttet til VAX, hvor det er ændret til at generere Pascal.

Når oversætteren er testet færdig, oversættes den med sig selv, og man er ikke længere afhængig af bootstrap værktøjet.

3.2.3 Systematisk oversætterkonstruktion

Grundlaget for konstruktion af oversætteren er en formel definition af Ada inklusive en abstrakt model af Ada tasking.

Den formelle definition er en denotationssemantik. En udmærket indføring heri findes i (Tennent 76) samt (Stoy 77) og (Stoy 80). Denne form muliggør en kompakt, omfattende, velgennemprøvet form for sprogdefinition. I de sidste år er der udviklet en metodik til systematisk udvikling af oversættere ud fra sådanne definitioner. (Bjørner 80) indeholder såvel bibliografi som en formel definition af væsentlige dele af Ada samt en A-kode.

I den formelle definition kan følgende dele identificeres:

- 1) Statisk semantik; denne definerer de dele af Ada, der kan kontrolleres under oversættelsen såsom synlighed af navne, behandling af erklæringer, typekontrol af parametre etc. Den statiske semantiks formler arbejder på en abstrakt form af Ada programmer, hvor programmerne er i overensstemmelse med en såkaldt abstrakt syntaks, AS1. Den statiske semantik afbilder AS1 programmer over i en anden abstrakt form kaldet AS2. I denne form er de dele af det oprindelige Ada program, der ikke er relevante for kodegenereringen, abstraheret bort (f.eks. erklæringer af exceptions). Desuden er visse konstruktioner blevet omformet, statiske udtryk blevet erstattet med deres værdi og symboltabelinformation blevet indsat.

Den første del af Front End oversætteren (scanneren og syntaksanalysatoren) konstrueres ved en kombination af 'håndkodning' og automatisk programgenerering. Resten, som udgør den mest komplekse del, konstrueres på grundlag af den statiske semantik.

Da formålet med den statiske semantik primært er at udgøre en del af definitionen af Ada, benytter den abstrakte datastrukturer (f.eks. afbildninger) og implicitte udtryk, som er kompakte, overskuelige og som passer til Ada-sproget. Men herved ligger formlerne ret langt fra en hensigtsmæssig implementering i f.eks. Ada. Det er derfor nødvendigt at ændre de abstrakte datastrukturer til nogle konkrete, der kan implementeres direkte, f.eks. lister og tabeller, og at ændre implicitte udtryk til algoritmer, der beregner de ønskede værdier.

DDC's Ada aktiviteter

- 2) Dynamisk semantik; denne giver semantikken - meningen - af Abstrakte Ada programmer. Den dynamiske semantik kan betragtes som en fortolker af abstrakte Ada programmer. Denne fortolker indeholder en model af lageret og af I/O systemet.

Fra den dynamiske semantik konstrueres oversætteralgoritmen, dvs. specifikationen af den kode, der skal genereres for hver Ada konstruktion, og fra denne selve kodegeneratoren.

- 3) Parallelisme model; denne del af den abstrakte fortolker håndterer processer (opstart, kommunikation mellem processer, fælles variable mellem processer, afslutning). Ud fra denne model afledes den del af køretidssystemet - kernen - som håndterer processer.

Det skal bemærkes, at strukturen af den statiske og den dynamiske semantik samt oversætteralgoritmen svarer til strukturen af Ada. Det er således nemt at finde semantikken af f.eks. en løkke sætning samt at se, hvilken A-kode den genererer.

DDC's Ada aktiviteter

3.3 Litteratur

- (Bjørner 80) D. Bjørner, O. Oest (eds.)
 "Towards a Formal Description of Ada"
 Lecture Notes in Computer Science, vol. 98
 Springer Verlag 1980.
- Indeholder:
- D. Bjørner, O. Oest: "The DDC Ada Compiler
 Development Project"
- J. Bundgaard, L. Schultz: "A Denotational
 (Static) Semantics Method for Defining Ada
 Context Conditions"
- J. Storbank Pedersen: "A Formal Semantics
 Definition of Sequential Ada"
- H.H. Løvengreen: "Parallelism in Ada"
- O. Dommergaard: "The Design of a Virtual
 Machine for Ada".
- (CR 83) Christian Røvsing A/S
 "A-Machine Specification"
 ADA/RFM/0001, issue 2
 1983-03-15
- (Stoy 77) J.E. Stoy
 "Denotational Semantics"
 MIT Press, Cambridge, Massachusetts, 1977.
- (Stoy 80) J.E. Stoy
 "Foundations of Mathematical Semantics"
 in "Abstract Software Specifications"
 (ed. D. Bjørner),
 Lecture Notes in Computer Science, vol. 86
 Springer Verlag 1980.
- (Tennent 76) R.D. Tennent
 "The Denotational Semantics of Programming
 Languages"
 Communications of the ACM, vol. 19, no. 8,
 August 1976.

Introduktion til Ada

4. Introduktion til Ada

Ada sproget forener mange af de sidste års fremskridt på programmeringssprogsområdet. Ada indeholder kun få begrebsdannelser som ikke kendes fra andre programmeringssprog, men Ada er det første sprog som på en elegant og simpel måde kombinerer disse begreber.

Sprog som Algol, PL/1, Simula og Pascal har dannet det væsentligste grundlag for designet af Ada. Sprog som Fortran, Basic, Cobol, Coral og APL blev anset for mindre interessante i forbindelse med designet af Ada.

De væsentligste nyskabelser i Ada i forhold til almindelig kendte programmeringssprog er følgende:

1. Kraftig typekontrol, der muliggør opdagelse af mange fejl allerede under oversættelsen af et program.
2. Dataabstraktion - Ada tillader indkapslede ("private") datatyper og dataspecifikationer som adskiller de abstrakte egenskaber ved data fra deres fysiske repræsentation.
3. Behandling af særlige situationer ("exceptions"), som giver programmøren mulighed for at kontrollere hvad der skal ske i fejlsituationer (overløb, division med nul osv.).
4. Parallel programmering ("tasking") - Ada er det første programmeringssprog med mulighed for større udbredelse, hvor muligheden for parallel programmering på naturlig måde er indbygget i sproget.
5. Modular programmering - mulighed for indkapsling af moduler og separat oversættelse, der ikke begrænser oversætterens kontrolmuligheder, f.eks. af underprogramkald.
6. Generiske moduler - mulighed for at parametrisere moduler med hensyn til data typer. Dette er af særlig interesse i forbindelse med konstruktion af programbiblioteker. Generiske moduler har visse lighedspunkter med makroer som kendes fra assembler-programmering.

I de følgende afsnit vil disse nyskabelser blive illustreret gennem en række simple programstumper skrevet i Ada. Af hensyn til læseligheden indgår bogstaverne æ, ø og å i en række af de anvendte identifikatorer. Dette er ikke tilladt ifølge Ada referencemanualen.

Introduktion til Ada

4.1 Simpelt eksempel

Vi indleder med et programeksempel, som giver en grov fornemmelse af den syntaktiske stil som et Ada program har.

Programmet, der hedder BABY, indlæser to tal, beregner deres sum, og udskriver resultatet. Hver programlinie indeholder en kommentar. Adas notation for kommentarer er '--' efterfulgt af kommentarteksten. Bemærk at Ada i et vist omfang er linieorienteret - der kan ikke stå programtekst efter kommentaren. Derimod kan der godt stå flere sætninger på samme linie, ligesom en sætning kan strække sig over flere linier.

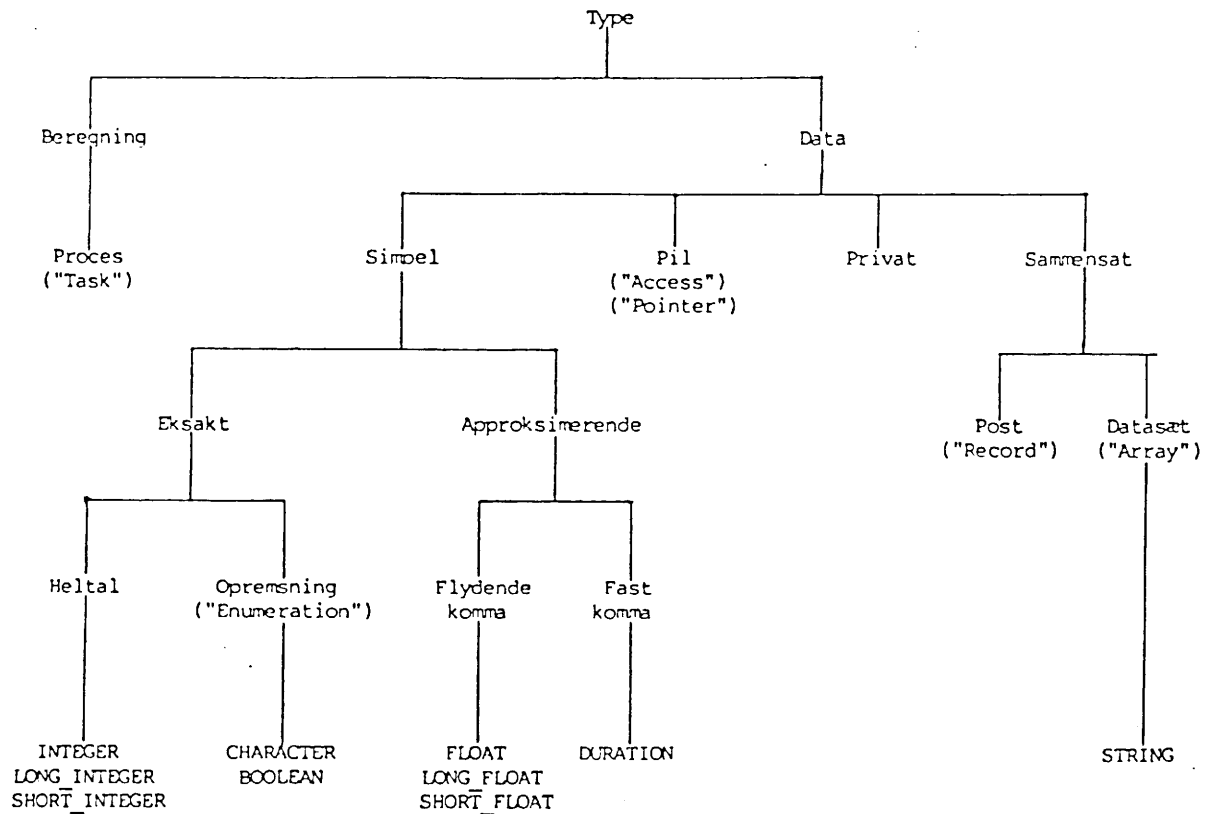
```
procedure BABY is
  A, B, SUM: INTEGER;
begin
  GET(A);  GET(B);
  SUM := A + B;
  PUT(SUM);
end BABY;
```

-- Erklæring af programmet BABY.
 -- Erklæring af 3 variable: A, B
 -- og SUM.
 -- BABY består af en række sætninger
 -- der indlæser A, og B,
 -- beregner summen af A og B
 -- og udskriver den.
 -- Slut på program BABY.

Introduktion til Ada

4.2 Datastrukturer

Ada giver programmøren mulighed for at arbejde med en lang række forskellige datatyper som vist i diagrammet nedenfor. Prædefinerede datatyper er vist med store bogstaver.



Introduktion til Ada

Hvert objekt ("variabel") i Ada sproget har en type, som fastlægger en række værdier, som dette objekt kan antage. Desuden fastlægger typen en række operationer, som kan udføres på objektet. Der findes fire slags datatyper: simple (skalare), sammensatte, pile og private.

Simple datatyper kan enten være numeriske eller brugerdefinerede.

Numeriske typer giver mulighed for at foretage eksakte eller approksimerende beregninger. Eksakte beregninger anvender heltalstypen, som står for sekvenser af heltal. Approksimerende beregninger anvender enten flydende komma typer hvor afrundingsfejlen er relativ, eller fast komma typer, hvor afrundingsfejlen er absolut.

Brugerdefinerede typer (opremsningstyper) består af en ordnet mængde af brugerdefinerede værdier. Den logiske type (BOOLEAN), tegnværdier (CHARACTER) er prædefinerede eksempler på opremsningstyper. Brugeren kan selv definere yderligere typer, som f.eks. farver, formulartyper, varer osv.

Sammensatte datatyper muliggør konstruktion af strukturerede objekter hvis enkelte komponenter er logisk beslægtede datatyper af vilkårlig art. De sammensatte datatyper i Ada er datasæt og poster. Et datasæt er et objekt der består af en række komponenter, der alle er af samme type, og som adresseres ved hjælp af et indeks. En post er et objekt med individuelt navngivne komponenter, der kan være af vilkårlig type. Det er muligt at definere poster med varianter. En sådan posts komponentstruktur afhænger af værdien af en af komponenterne, den såkaldte diskriminant.

Pile muliggør dynamisk konstruktion af sammenkædede datastrukturer (lister, træer) ved hjælp af standardfunktioner. Flere variable af piltypen kan udpege det samme objekt og delkomponenter i et sammensat objekt kan udpege det samme eller andre objekter. De enkelte elementer i en sådan sammenkædet datastruktur kan ændres dynamisk, ligesom deres indbyrdes relationer kan ændres af det kørende program.

I en programpakke er det muligt at definere private datatyper, dvs. datatyper hvis eksakte struktur kun er delvis tilgængelig (evt. helt utilgængelig) uden for programpakken. Kun logisk relevante dele inklusive eventuelle diskriminanter defineres som værende tilgængelige uden

Introduktion til Ada

for programpakken. Private typer er nyttige hvis en konstruktør af en standard programpakke ønsker at bevare en vis frihed med hensyn til senere ændringer i programpakkens interne struktur, idet han kan skabe en veldefineret grænseflade til sine brugere.

En programmør kan definere sine egne datatyper, hvilket muliggør fejlkontrol og dataabstraktion:

declare

```
type  FORMAT is  (STANDARD,A4HØJ,A4TVÆR,A3,ETIKETTE);
```

```
type  TRYKFARVE is  (BLÅ, SORT, RØD, GRØN, GRÅ);
```

```
N: TRYKFARVE;           -- N er en variabel af type TRYKFARVE
M: FORMAT;              -- M er en variabel af type FORMAT
FAKTURA: FORMAT:=STANDARD; -- erklæring og initialisering
FAKTURAFARVE: TRYKFARVE;  -- erklæring uden initialisering
```

begin

```
N := GRØN;              -- ok
FAKTURAFARVE := N;      -- ok
N := A4HØJ;             -- fejl: typeuoverensstemmelse
```

end;

Ovenstående eksempel kan naturligvis også realiseres i andre programmeringssprog, f.eks. ved at benytte heltal som repræsentation for data (STANDARD=1, A4HØJ=2, ... , ETIKETTE=5, BLÅ=1, SORT=2, osv.). Fordelen ved Ada er at programmøren ikke bebyrdes med irrelevante overvejelser vedrørende repræsentationen af data.

Desuden er en effektiv fejlkontrol mulig som vist i den fejlbehæftede tildelingssætning hvor værdien forsøges tildelt N i stedet for M. I ældre programmeringssprog ville oversætteren ikke have haft mulighed for at opdage den tilsvarende fejl i sætningen N:=2. Fejlen ville derfor have været langt mere besværlig at finde.

Introduktion til Ada

Det er muligt at opbygge poster (sammensatte datatyper, "records"). Det følgende eksempel viser opbygningen af en simpel post, som kan rumme en dato. Man ser at programmøren ikke behøver at bekymre sig om den interne repræsentation af data i posten. Hele posten kan initialiseres i én operation.

declare

```
type MÅNEDSNAVN is (JAN, FEB, MAR, APR, MAJ, JUN,
                     JUL, AUG, SEP, OKT, NOV, DEC);
```

```
type DATO is
  record
    DAG:  INTEGER range 1..31;
    MÅNED: MÅNEDSNAVN;
    ÅR:   INTEGER;
  end record;
```

```
DDAG:  DATO := (6, JUN, 1944);
TERMIN: DATO;
```

begin

```
  TERMIN      := (11, JUN, 1981);
  TERMIN      := ( ÅR=>1981, MÅNED=>JUN, DAG=>11 );
  TERMIN.MÅNED := DEC;
```

end;

Det er muligt i begrænset omfang at parametrisere poster, dvs. at definere en posttype således at de enkelte konkrete poster af denne type i begrænset omfang kan afvige fra hinanden i struktur (postvarianter):

declare

```
type MK is ( M, K );
```

```
type PERSON( KØN: MK ) is
  record
```

```
    FØDT: DATO; -- forudsætter foregående erklæring af DATO
    case KØN is
      when M => MODIG, AFTJENT_VÆRNEPLIGT: BOOLEAN;
      when K => ANTAL_BØRN: INTEGER range 0..30;
    end case;
  end record;
```

```
type KARTOTEK is array(INTEGER range 1..999) of PERSON;
```

```
OLFERT: PERSON := ( M, (17, JUN, 1959), FALSE, FALSE );
TRINE:  PERSON := ( K, (20, APR, 1902), 9 );
```

Introduktion til Ada

Modsat Pascal er det muligt at definere arrays med variable grænser i underprogrammer og blokke:

```
procedure OPRET_KARTOTEK( ANTAL_MEDLEMMER: INTEGER ) is

  type MEDLEM is
    record
      NAVN, ADRESSE, BY: STRING(30);
      KLASSE: (ATOM,MYG,FLUE,BANTAM,LET,MELLEM,SVÆR,SUPER);
    end record;

  MEDLEMSKARTOTEK: array( 1..ANTAL_MEDLEMMER ) of MEDLEM;

begin
  for I in MEDLEMSKARTOTEK'RANGE loop
    INDLÆS( MEDLEMSKARTOTEK(I) );
  end loop;
  WRITE( BÅND, MEDLEMSKARTOTEK ); -- udskriv kartotek
end; -- MEDLEMSKARTOTEK nedlægges når proceduren forlades
```

Introduktion til Ada

Man kan også arbejde med tegnstreng. Modsat Pascal indeholder Ada fuldt tilfredsstillende muligheder for manipulation af tegnstreng. Ada sproget bygger i øvrigt på så simple og kraftfulde begreber at tegnstreng kan defineres inden for rammerne af Ada sproget selv, hvilket er gjort i standardpakken ASCII:

```
type CHARACTER is (NUL,SOH, .. , 'A','B', .. , 'a', .. , DEL);
subtype NATURAL is INTEGER range 1..INTEGER'LAST;
type STRING is array (NATURAL range <>) of CHARACTER;
```

Indeksgrænserne for en STRING opgives ikke eksplicit i typedefinitionen og skal derfor fremgå af erklæringen, enten eksplicit eller implicit, f.eks. ved at initialisere tegnstrengen med en passende værdi:

```
MATADOR: constant STRING := "Av min arm";    -- længde 10 tegn
LINIEBUFFER: STRING(80);                      -- længde 80 tegn
ADVARSEL: constant STRING := "Pas på!" & ASCII.BEL&ASCII.BEL;
SKILLELINIE: constant STRING(1..120) := (1..120 => '-');
```

Læsning og skrivning i tegnstreng:

```
A( 1..10 ) := "Far kommer";
A( 4..13 ) := A( 1..10 );    -- A( 1..13 ) = "FarFar kommer"
```

Værdien af en funktion kan være en sammensat datatype, f.eks. en tegnstreng:

```
function OMBYT( NAVN: STRING ) return STRING is
begin    -- (fornavn efternavn) ændres til (efternavn,fornavn)
    for INDEX in reverse NAVN'FIRST..NAVN'LAST-1 loop
        if INDEX = NAVN'FIRST then return NAVN; end if;
        if NAVN( INDEX ) = ' ' then
            return NAVN( INDEX+1..NAVN'LAST ) & ',' &
                NAVN( NAVN'FIRST..INDEX-1 );
        end if;
    end loop;
end;
```

Introduktion til Ada

4.3 Kontrolstrukturer

Der er følgende sætningstyper i Ada:

Simple:

- tildeling
- procedure- eller funktionskald. Herunder hører al ind/uddata, der udføres af prædefinerede biblioteksunderprogrammer
- udhop af løkke (exit)
- ubetinget hop (goto)
- udhop af funktion/procedure (return)
- den tomme sætning (null)

Sammensatte:

- blok (declare ... begin ... exception ... end)
- valg (if ... then ... else ... end if)
- forgrening (case ... when ... end case)
- løkker
 - uendelige (loop ... end loop)
 - med betinget udhop (loop ... exit ... end loop)
 - fast antal gennemløb (for ... loop ... end loop)
 - iteration (while ... loop ... end loop)

Behandling af fejlsituationer ("exceptions"):

- angivelse af fejlsituation (raise)
- blok til behandling af fejlsituation ("exception handler", exception ... when ... end)

Parallel afvikling ("tasking")

- send meddelelse (bemærk: der er ikke tale om meddelelser i traditionel forstand)
- modtag meddelelse (accept ... do ... end)
- modtag en af flere mulige meddelelser (select)
- tidsforsinkelse (delay)
- stop proces (abort)

Sætninger skrevet i den underliggende maskines sprog.

Introduktion til Ada

Eksempler:

Simpelt valg:

```
if A > MAX then
  MAX := A;
end if;
```

Valgsætning med alternativ:

```
if SVAR = 'J' then
  REGISTRER_TRANSAKTION;
  UDSKRIV_KVITTERING;
  KUNDE.SALDO := TRANSAKTION.BELØB + KUNDE.SALDO;
else
  PUT( "Transaktion afvist" );
end if;
```

Valgsætning med foreskreven beregningsrækkefølge for de logiske udtryk. Det 3. logiske udtryk må kun beregnes hvis både det første og det andet logiske udtryk er sande. Beregningsrækkefølgen for det første og andet logiske udtryk er derimod ikke foreskrevet:

```
if (KUNDENR > 0 and
    KUNDENR <= SIDSTE_KUNDE) and then
  KARTOTEK( KUNDENR ).STATUS /= UDGÅET then
  UDSKRIV_ADRESSEMÆRKAT;
end if;
```

Forgreningssætningen (case) tillader et valg mellem en af flere muligheder:

```
type UGEDAG is (MANDAG, TIRSDAG, ONSDAG, TORSDAG,
  FREDAG, LØRDAG, SØNDAG, UKENDT);
IDAG: UGEDAG;
.
.
.
case IDAG is
  when MANDAG => NULSTIL_REGNSKAB;
  when TIRSDAG..TORSDAG => UDSKRIV_DAGSRAPPORT;
  when FREDAG => AFSLUT_REGNSKAB;
  when LØRDAG|SØNDAG => null; -- ingen aktion
  when others => PUT( "Dato ikke indtastet" );
end case;
```

Introduktion til Ada

Der findes tre forskellige former for løkker, der alle er variationer over samme tema:

LÆSESLØJFE:

```
loop
  LÆS_DATA( TEGN );
  case TEGN is
    when 'A'      => UDFØR_AKTION;
    when 'X'      => exit LÆSESLØJFE;
    when others   => PUT( "Indtastningsfejl." );
  end case;
end loop LÆSESLØJFE;
```

Oversætteren kontrollerer at det navn der evt. står efter end loop modsvarer det navn der kan stå foran løkken. Andre kommentarer til den afsluttende sætning skal skrives på sædvanlig vis.

```
for I in A'RANGE loop
  A(I) := 0;
end loop;
```

```
while NEXT /= null loop
  SUM := SUM + NEXT.VALUE;
  NEXT := NEXT.SUCC;
end loop;
```

Introduktion til Ada

Udførelsen af et program kan føre til fejlsituationer, som bevirker at normal sekventiel udførelse af programmet ikke kan fortsætte. Eksempelvis kan resultatet af en aritmetisk beregning være større end den største tilladte numeriske værdi i datamaten (overløb) eller et fejlbehæftet program kan forsøge at anvende et ukorrekt indeks i en datasæt operation. Med henblik på sådanne situationer giver Ada programmøren mulighed for ved hjælp af såkaldte "exception handlers" at specificere hvilke aktioner en fejlsituation af en bestemt type skal medføre.

Foruden de prædefinerede fejltyper har programmøren mulighed for at definere sine egne fejltyper. En exception handler kan specificere forskellige aktioner for forskellige fejltyper. Fejlsituationer kan signaleres eksplicit ved hjælp af en raise sætning.

Eksempel:

```
function FACTORIAL( N: NATURAL ) return FLOAT is
begin
  if N = 1 then
    return 1.0;
  else
    return FLOAT( N ) * FACTORIAL( N-1 );
  end if;
exception
  when NUMERIC_ERROR => return FLOAT'LARGE;
end FACTORIAL;
```

Introduktion til Ada

4.4 Programstruktur

I forbindelse med Adas programstruktur benytter man følgende begreber:

- Et underprogram ("subprogram") anvendes til at beskrive en algoritme eller en afgrænset del heraf. Ada har to slags underprogrammer: procedurer og funktioner. En procedure er logisk set ækvivalent med en serie aktioner. En funktion er logisk set ækvivalent med beregning af en værdi. Funktioner og procedurer har mange fællestræk, men en funktion vil altid returnere en værdi.

- En pakke ("package") anvendes til at beskrive en samling logisk beslægtede programenheder. Eksempelvis kan en pakke beskrive et fælles sæt af erklæringer (typer/ variabelærklæringer), en samling beslægtede underprogrammer, eller et sæt typeerklæringer med tilhørende operationer i form af underprogrammer. Dele af en pakke kan skjules for brugeren ("private") hvorved designeren af pakken sikres fuld kontrol over hvilke dele af pakken der udgør snitfladen til omverdenen.

Eksempler: pakke til ind/uddataoperationer, styring af grafisk farveskærm eller matrixberegninger.

- En proces ("task") anvendes til at beskrive aktionssekvenser som kan udføres parallelt med andre processer. En procesbeskrivelse kan definere enten en enkelt proces eller en familie af identiske processer.

Eksempler: udskrivning af rapport kan udføres parallelt med indtastning af nye data; et styreprogram for en dataskærm kan startes op en gang for hver terminal i systemet.

- Moduler ("compilation units"), som er programdele der kan oversættes separat, dvs. uden at resten af det programsystem, de tilhører, oversættes samtidig. Separat oversættelse medfører ikke, at oversætterens kontrol af programmet slækkes, idet procedureparametre og korrekt brug af datatyper kontrolleres som om den øvrige programtekst blev oversat sammen med modulet.

Eksempel: ved udvikling af større programsystemer opdeles udviklingen i moduler; afprøvning af nye moduler kræver ikke fornyet oversættelse af eksisterende moduler.

Introduktion til Ada

Eksempel på procedurer i Ada:

```

type ASCIIITAL is STRING(4);

function ASCII_TIL_BINÆR( INDDATA:ASCIIITAL ) return INTEGER is
    RESULTAT: INTEGER := 0;

begin
    for I in reverse INDDATA'RANGE loop
        RESULTAT := RESULTAT*10 + ( CHARACTER'POS(INDDATA(I))
                                   - CHARACTER'POS('0') );
    end loop;
    return RESULTAT;
end ASCII_TIL_BINÆR;

```

Når man bestiller en martini i USA står man over for et valg der illustreres af følgende typebeskrivelser:

```

type SPIRIT is ( GIN, VODKA );
type STYLE is ( ON_THE_ROCKS, STRAIGHT_UP );
type TRIMMING is ( OLIVE, TWIST );

```

Når man definerer bartenderproceduren er det muligt at specificere standardværdier, f.eks.

```

procedure DRY_MARTINI( BASE: SPIRIT := GIN;
                      HOW: STYLE := ON_THE_ROCKS;
                      AVEC: TRIMMING := OLIVE );

```

Bartenderproceduren kan nu f.eks. kaldes på følgende måder

```

DRY_MARTINI;

DRY_MARTINI( VODKA, AVEC => TWIST );

DRY_MARTINI( HOW => STRAIGHT_UP );

```

Introduktion til Ada

4.5 Afsluttende eksempel

Det følgende eksempel, der er hentet fra Ada reference-manualen, giver et udmærket indtryk af slagkraften bag de forskellige konstruktioner i Ada.

```

with TEXT_IO; use TEXT_IO;
procedure DIALOG is
  type FARVE is (HVID, RØD, ORANGE, GUL, GRØN, BLÅ, BRUN);
  package FARVE_IO is new ENUMERATION_IO( ENUM=>FARVE );
  package HELTAL_IO is new INTEGER_IO(INTEGER);
  use FARVE_IO, HELTAL_IO;

  LAGER: array(FARVE) of INTEGER:=(20,17,43,10,28,173,87);
  VALG: FARVE;

  procedure INDFAST_FARVE(VALGT_FARVE: out FARVE) is
    begin
      loop
        begin
          PUT( "Indtast farve: "); -- spørg brugeren
          GET( VALGT_FARVE );
          return;
        exception
          -- hvis der indtastes ukendt farve
          when DATA_ERROR =>
            PUT( "Ukendt farve - prøv igen. ");
            NEW_LINE(2);
        end;
      end loop; -- fortsæt indtil lovlig farve indtastes
    end;

begin
  -- sætninger i procedure DIALOG

  HELTAL_IO.DEFAULT_WIDTH := 5; -- fastlæg standardbredde
  loop
    INDFAST_FARVE(VALG);
    SET_COL(5);
    PUT( "Antal " ); PUT( VALG ); PUT( " på lager: " );
    SET_COL( 25 ); PUT(LAGER(VALG)); -- standardbredde 5
    NEW_LINE;
  end loop;

end DIALOG;

```

Eksempel på dialog mellem bruger og program (oplysninger indtastet af brugeren er vist i *kursiv*):

Indtast farve: *sort*
 Ukendt farve - prøv igen.

Indtast farve: *blå*
 Antal blå på lager: 173;

Introduktion til Ada

4.6 Afledte egenskaber

Ada sproget er designet med henblik på en række egenskaber der er af væsentlig betydning ved effektiv programmeludvikling. Disse egenskaber er

- Modular programmeludvikling. Oversætteren understøtter effektivt moderne programmeludviklingsmetoder, såsom bottom-up og top-down metoderne.
- Generel anvendelighed. Ada sproget er designet med henblik på indlejrede datamatsystemer, og er derfor særdeles velegnet til alle opgaver inden for proceskontrol. Sproget synes imidlertid også at være fuldt på højde med de bedste sprog til videnskabelig og administrativ databehandling (talbehandling, tegnbehandling, biblioteksfunktioner, effektivitet). Herudover er sproget velegnet til udvikling af styresystemer (multiprogrammeringsfaciliteter).
- Sikkerhed. Kraftig typekontrol og effektiv dataabstraktion vil medføre opdagelse af mange fejl allerede under oversættelsen af et program.
- Effektivitet. Ada sproget er designet med henblik på effektiv oversættelse og mulighed for generering af effektiv kode, både hvad angår plads og effektivitet.
- Standardisering. Ada sproget vil blive standardiseret af ANSI og formodentlig også af ISO. Belært af erfaringerne fra Fortran og Cobol vil oversættere kun kunne føre betegnelsen Ada oversætter hvis de netop kan oversætte det sprog der er beskrevet i Ada referencemanualen. Udvidelser og arbitrære delmængder vil ikke blive tilladt. Derimod vil der kunne forekomme Ada oversættere der ikke understøtter visse faciliteter i sproget, som f.eks. flydende tal og parallel programmering.

Omgivelserne, dvs. grænsefladen til styresystemet forventes også at blive standardiseret.

Standardiseringen vil give anledning til fremkomsten af en lang række lærebøger og kurser i Ada, ligesom de fleste læreanstalter vil indføre undervisning i Ada.

Introduktion til Ada

- Flytbarhed. Den effektive standardisering vil gøre det muligt at flytte Ada programmer fra en installation til en anden uden ændringer. Standardiseringen vil yderligere medføre en opblomstring af programmelindustrien som får et væsentligt større marked for sine Ada produkter.

Oplæringstiden for Ada programmører ved overgang til en ny maskintype vil blive væsentlig formindsket.

- Stabilitet. Ændringer i standarden for Ada sproget vil tidligst blive tilladt i 1988.

Introduktion til Ada

4.7 Sammenligning mellem Ada og andre programmeringssprog

	Fortran	Cobol	PL/1	Pascal	Ada
Programstruktur	+	+	+++	++	+++
Parallelle processer ("Tasking")	-	-	+	-	++
Separat oversættelse	++	++	++	-	+++
Maskinnær programmering	+	+	+	-	+++
Kontrolstruktur	+	+	++	+++	+++
I/O faciliteter	++	++	++	+	++
Videnskabelige beregningsopgaver	++	-	+++	++	+++
Administrative beregningsopgaver	-	+++	+++	+	++
Datastrukturer	-	++	++	++	+++
Typeapparat	+	+	+	++	+++
Brugerdefinerede typer	-	-	-	++	+++
Undervisningsmateriale	+++	+++	++	++	+
Standardisering	++	++	++	+	+++
Flytbarhed	++	++	+	+	+++
Effektivitet (udførelse)	++	++	+	++	++
Sikkerhed mod programfejl	-	-	+	++	+++

Parallel programmering i Ada

5. Parallel programmering i Ada

Lad os forestille os en datamaskine, til hvilken der er koblet flere terminaler. Der kræves i denne maskine et program til at styre kommunikationen mellem en bruger ved en terminal og datamaskinen. På den anden side er det også klart, at alle terminalerne kan betjenes af identiske programmer. Disse bør naturligvis køre parallelt, sådan at alle terminaler kan bruges samtidigt.

Man kunne tænke sig, at der blev skrevet ét sådant program, og at der derefter oprettes et antal processer (svarende til antallet af terminaler), som alle udfører dette program. Dette ene program kan skrives, som om der kun er én terminal på systemet, og skal altså ikke løse det meget komplicerede problem med at betjene flere terminaler samtidigt.

Hvis vi skal løse et sådant problem har vi stor nytte af parallel programmering. Der kunne nævnes mange andre eksempler på anvendelser af parallel programmering indenfor områder som f.eks. processtyring, simulering og systemprogrammering.

I Ada har man i modsætning til de fleste andre alment udbredte programmeringssprog mulighed for at anvende parallel programmering i form af processer ("tasks").

Selv om man logisk set kan betragte to processer som kørende parallelt, behøver dette dog ikke fysisk at finde sted. Et og samme Ada program indeholdende flere processer vil kunne afvikles ægte parallelt (på flere maskiner i et datamatnetværk eller på flere centralenheder i samme maskine) eller kun logisk set parallelt (tidsdeling på en enkelt centralenhed) eller som en kombination af disse muligheder.

En proces introduceres i et Ada program i to omgange. Først kommer specifikationen, som indeholder en beskrivelse af den måde, hvorpå der kan kommunikeres med processen:

```
task T is
.
.
end;
```

Derefter følger selve kroppen, som beskriver de aktioner processen skal udføre:

```
task body T is
.
.
end T;
```

Parallel programmering i Ada

Eksempel :

Vi tænker os et lysreguleret vejkryds. I dette vejkryds befinder sig en bil. Lyset skifter uafhængigt af bilistens opførsel, og bilen kan principielt også køre ganske uafhængigt af lyset. Vi har altså to parallelle processer. Simuleringen af dette vejkryds kan altså groft skitseres på følgende måde:

```
procedure TRAFIK is

  task BIL is
    .
    .
    .
  end;

  task TRAFIKLYS is
    .
    .
    .
  end;

  task body BIL is
    .
    .
    .
  end BIL;

  task body TRAFIKLYS is
    .
    .
    .
  end TRAFIKLYS;

begin
  .
  .
  .
end TRAFIK;
```

Her vil BIL, TRAFIKLYS og selve proceduren TRAFIK udvikles parallelt. BIL og TRAFIKLYS stopper, når de når til deres sidste end.

Indtil nu har vi kun talt om processer som helt selvstændige individer uden nogen form for kontakt. De kunne lige så godt have været adskilt fuldstændigt i separate programmer. Vi vil nu se på, hvordan processer kan kommunikere med hinanden.

Parallel programmering i Ada

5.1 Kommunikation og synkronisering

Selv om et trafiklys og en bil er selvstændige individer, har de brug for at kommunikere under en eller anden form. Når lyset bliver rødt, bør bilen stoppe, når lyset bliver grønt, bør den køre videre. Skærmterminalerne har også brug for at kunne kommunikere, sådan at der kan sendes meddelelser fra skærm til skærm.

Kommunikationen mellem processer foregår i Ada ved hjælp af et såkaldt entry. På dansk kunne man kalde dette et synkroniseringspunkt. En proces definerer i sin specifikation de tilbudte entries :

```
task BIL is
```

```
    entry STOP;  
    entry KØR(X : in RETNING);
```

```
end;
```

Der er her defineret en proces som har to entries. Tanken bag ved netop dette valg er naturligvis, at TRAFIKLYS skal kunne bede BIL om at stoppe og om at køre (i en eller anden retning) i takt med at lyset skifter i lyskurven.

De aktuelle aktioner i forbindelse med synkroniseringspunkterne beskrives ved hjælp af accept sætninger :

```
task body BIL is
```

```
    .  
    .
```

```
    begin
```

```
        loop
```

```
            accept STOP do
```

```
                .
```

```
                .
```

```
                .
```

```
            end;
```

```
            accept KØR(X : in RETNING) do
```

```
                .
```

```
                .
```

```
                .
```

```
            end;
```

```
        end loop;
```

```
end BIL;
```

Parallel programmering i Ada

BIL accepterer at "blive stoppet" og derefter skiftevis at "blive startet" og "stoppet". Kald til de to entries ser sådan ud:

```
BIL.STOP;
```

```
BIL.KØR(RETN);
```

hvor RETN er en variabel eller konstant af typen RETNING. Det tænkes her at variable af typen RETNING kan antage værdierne FREMAD, TIL_HØJRE, TIL_VENSTRE eller ALLE_RETNINGER.

Bemærk, at et entry-kald syntaktisk set er identisk med et procedure-kald. Den kaldende proces behøver altså ikke at vide, om den kalder en procedure eller et entry.

I tilfældet STOP overføres ingen data. Vi siger, at der er tale om synkronisering. I tilfældet KØR derimod sker en overførsel af data (retningen). Her taler vi om kommunikation. BIL får at vide, hvilken retning det nu er tilladt at køre i. Hvis denne retning stemmer overens med bilens eget ønske, må den køre, ellers ikke. Der er dog ingen garanti for, at den vil opføre sig lovligt.

For at to processer skal kunne kommunikere kræves, at de foretager et såkaldt rendezvous. Et rendezvous opnås, når en proces kalder et entry i en anden, og denne anden er klar til at blive kaldt, d.v.s. til at udføre en accept. Begge processer kan altså blive nødt til at vente på, at et rendezvous skal foregå. Den kaldende skal eventuelt vente indtil den anden proces bliver klar, eller denne skal vente på at blive kaldt. Under rendezvous-et bliver den "fælles" kode i accept sætningen udført. Derefter kører begge processer videre uafhængigt af hinanden. Ved et rendezvous opnås, at begge processer befinder sig i en veldefineret tilstand. Bemærk iøvrigt, at et rendezvous altid medfører en synkronisering uafhængigt af, om der overføres data.

Betragt følgende to udgaver af en accept sætning :

```
accept TÆL do
  X:=X+1;
end;
```

```
accept TÆL;
  X:=X+1;
```

hvor X er en global variabel.

En kaldende proces kunne udføre følgende :

```
X := 0;
TÆL(X);
PUT(X);
```

Parallel programmering i Ada

Hvis TÆL er udformet som i det første tilfælde, vil det udskrivne resultat altid være 1, fordi optællingen foregår under rendezvous. I det andet tilfælde kan resultatet blive både 0 og 1, fordi optællingen først foregår efter rendezvous. Heraf ses tydeligt nødvendigheden af at kunne udføre kode i den veldefinerede tilstand, som et rendezvous giver.

Det ses af ovenstående at rendezvous-mekanismen er asymmetrisk. Den kaldende proces skal kende navnet på den, der kaldes. Sidstnævnte derimod skal blot være i stand til at blive kaldt af en hvilken som helst anden proces.

Som følge af dette vil ethvert entry kunne have ikke blot én proces, men en hel kø af sådanne ventende på rendezvous. Disse ventende processer vil blive betjent i samme rækkefølge, som de udførte forsøg på kald af det pågældende entry.

I foregående eksempel havde vi en proces (BIL), som kun ville acceptere at ét entry ad gangen blev kaldt. Dette er faktisk ret uheldigt. Lad os forstille os, at trafiklyset er i én af tre mulige tilstande :

- 1: rød.
- 2: rød men med grøn højre-pil.
- 3: grøn.

Der skiftes i denne rækkefølge : 1-2-3-1...

Lad os videre forestille os at bilen vil lige ud. Ved skiftet fra 1 til 2 får bilen besked på, at det er tilladt at køre til højre. Da den imidlertid ikke vil til højre, foretager den sig ikke videre, men hopper ud i loop og tilbage til accept STOP. Den når altså aldrig at komme ud i krydset.

Dette problem kunne løses på flere måder. Bilen kunne f.eks. drage nytte af, at det grønne lys kommer lige efter pilen, og således sige accept KØR to gange i træk. En bedre løsning, hvor bilen ikke udnytter noget kendskab til lysets opførsel, er at bruge den såkaldte select sætning. Ved hjælp af denne er det muligt at vente på flere begivenheder samtidigt, som vist i nedenstående eksempel.

Parallel programmering i Ada

```

task body BIL is
.
.
.
begin
    loop
        select
            accept STOP do
                .
                .
                .
            end;
        or
            accept KØR(X : in RETNING) do
                .
                .
                .
            end;
        end select;
    end loop;
end BIL;

```

Her vil både STOP og KØR kunne accepteres, efter at BIL har begyndt udførelsen af select sætningen. Det kan altså nu lade sig gøre at foretage flere kald af f.eks. KØR uden at STOP bliver kaldt.

I alle hidtil viste eksempler optrådte en given proces kun i ét eksemplar. Det er i Ada muligt at definere såkaldte task typer og ved hjælp af en sådan erklære flere objekter af samme task type, sådan at vi har adskillige eksemplarer af samme proces kørende parallelt.

Dette svarer til at vi i vores vejkryds havde adskillige biler og måske også to trafiklys (et for hver retning). Alle biler vil være af samme type, men dog have hver sin eksistens.

Det er ydermere muligt i Ada under selve programudførelsen (dynamisk) at oprette flere eksemplarer af samme TASK. En udvidelse af trafikeksemplet ville være at have endnu en proces, som løbende "skaber" biler og sender disse ud i trafikken. Der vil her ikke blive gjort nærmere rede for dynamisk oprettelse eller nedlæggelse af processer (se Ada reference manualen).

Parallel programmering i Ada

Vi vil nu se på et mere komplet eksempel på proces-kommunikation. Eksemplet er hentet fra Ada reference manualen.

Vi har en proces, som producerer tegn, f.eks. fra en terminal. Disse tegn bliver gemt i en buffer, hvorfra en anden proces, der f.eks. styrer en linieskriver, kan hente dem. Til at styre bufferen til lagring af tegn har vi en tredje proces kaldet BUFFER. Denne har to indgange (entries): SKRIV_TEGN og LÆS_TEGN.

Den producerende proces kan indeholde følgende sætninger:

```
loop
  GET(TEGN);    -- producer det næste tegn i variabelen TEGN
  BUFFER.SKRIV_TEGN(TEGN);
  exit when TEGN = SLUT_TEGN;
end loop;
```

Den modtagende proces kan indeholde følgende :

```
loop
  BUFFER.LÆS_TEGN(TEGN);
  PUT(TEGN);    -- brug tegnet
  exit when TEGN = SLUT_TEGN;
end loop;
```

Parallel programmering i Ada

BUFFER styrer en pulje af tegn. Når denne pulje ikke er fyldt tillades skrivning. Når den ikke er tom, tillades læsning. Når en skrivning har fyldt sidste plads i puljen, vil næste skrivning foregå på første plads, idet vi kan være sikre på at det tegn, der står her i forvejen allerede er blevet læst.

task BUFFER is

entry LÆS_TEGN(T : out CHARACTER);
 entry SKRIV_TEGN(T : in CHARACTER);

end;

task body BUFFER is

 MAX_ANTAL : constant INTEGER := 100;
 PULJE : array(1..MAX_ANTAL) of CHARACTER;
 ANTAL : INTEGER range 0..MAX_ANTAL := 0;
 I_IND, I_UD : INTEGER range 1..MAX_ANTAL := 1;
begin

loop

select

when ANTAL < MAX_ANTAL =>
 accept SKRIV_TEGN(T : in CHARACTER) do
 PULJE(I_IND) := T;
 end;
 I_IND := I_IND mod MAX_ANTAL + 1;
 ANTAL := ANTAL + 1;

or when ANTAL > 0 =>
 accept LÆS_TEGN(T : out CHARACTER) do
 T := PULJE(I_UD);
 end;
 I_UD := I_UD mod MAX_ANTAL + 1;
 ANTAL := ANTAL - 1;

or terminate; -- stop når producerende og modtagende
 -- proces er stoppet

end select;

end loop;

end BUFFER;

Parallel programmering i Ada

5.2 Tidsforsinkelser.

Der kan forekomme situationer, hvor en proces ønsker at vente et vist tidsrum, inden den kører videre. Ligesådan kan der være situationer, hvor en proces kun ønsker at vente i et vist maksimalt tidsrum på at opnå rendezvous. Til dette formål findes DELAY sætningen. En proces, der ønsker at vente i 2 sekunder kan udføre sætningen:

```
delay 2.0;
```

En utålmodig bilist, der højst vil vente 1½ minut på, at der bliver grønt lys, og som kører over for rødt, hvis ikke det sker, kan simuleres gennem sætningen :

```
select
```

```
    accept KØR(X : in RETNING) do
    .
    .
    .
    end;
```

```
or
```

```
    delay 90.0;
```

```
end select;
```

Et trafiklys som kun styrer én bil (ved navn BIL), og som har følgende intervaller :

```
    rødt lys (incl. efterfølgende gult lys): 60 sek.
    grøn højrepil : 15 sek.
    grønt lys (incl. efterfølgende gult lys): 45 sek.
```

kan simuleres gennem sætningen :

```
loop
```

```
    BIL.STOP;
    delay 60.0;
    BIL.KØR(TIL_HØJRE); -- tilladelse til at køre til højre
    delay 15.0;
    BIL.KØR(ALLE_RETNINGER); -- generel kørselstilladelse
    delay 45.0;
end loop;
```

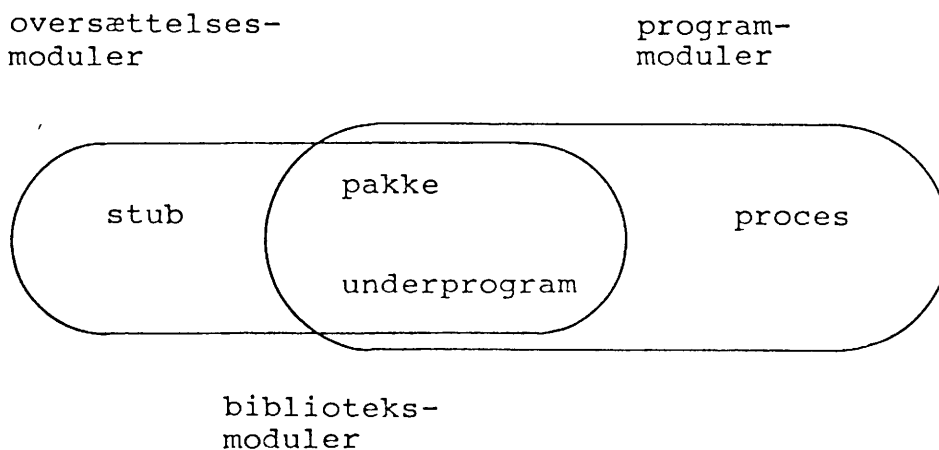
Modulær programmering

6. Modulær programmering i Ada

En af fordelene ved Ada er, at der i modsætning til f. eks. Pascal og Algol er defineret programstrukturer til modulær programmering, og at det er muligt at oversætte programstrukturer separat uden tab af sikkerhed. Dette skyldes, at oversætteren kan udføre den samme grad af kontrol hvadenten et Ada program har form af én programstruktur eller af flere, der oversættes hver for sig. Reglerne for separat oversættelse indgår i definitionen af sproget. Det er muligt at opbygge biblioteker af programstrukturer. Det er også muligt at flytte sådanne biblioteker til nye Ada systemer takket være ovennævnte standardisering.

Der findes tre programstrukturer, som benyttes ved modulær programmering: Pakker ("packages"), underprogrammer ("subprograms") samt processer ("tasks"). Kun de to første kan oversættes separat. Yderligere kan såkaldte stubbe ("stubs"), se senere, oversættes separat. Det er derfor nødvendigt at skelne mellem

- programmoduler ("program units"), som er de programstrukturer, der kan benyttes til modulær programmering
- oversættelsesmoduler ("compilation units"), som er de programstrukturer, der kan oversættes separat
- biblioteksmoduler ("library units"), som er de programstrukturer, der kan indgå i biblioteker.



Processer kan således ikke oversættes separat, men da processer kan indgå i pakker, kan processer alligevel indgå i et programbibliotek.

Modulær programmering

6.1 Programmoduler

Disse udgøres af pakker, underprogrammer (procedurer og funktioner) og processer. De vil normalt bestå af en specifikationsdel og en krop. Specifikationsdelen indeholder den information, som er nødvendig, når man skal benytte programmodulet. Det vil for underprogrammer f. eks. sige navn og type af parametre, evt. standard ("default") værdier, og om parametrene overføres som værdi, værdi/resultat eller resultatparametre (in, in out eller out). Funktionen af underprogrammet må angives i en kommentar. Da specifikationsdelen udgør hele grænsefladen til omverdenen, kan selve implementeringen af kroppen ændres uden at de programmer, der benytter modulet, skal ændres.

Lad os som eksempel på programmoduler betragte pakker. I det første eksempel består modulet kun af en specifikationsdel, da det kun stiller nogle datatyper og objekter til rådighed, men ingen operationer:

```
package ARBEJDS_DATA is

  type DAG is (MANDAG, TIRSDAG, ONSDAG, TORSDAG,
               FREDAG, LØRDAG, SØNDAG);

  type TIMEFORBRUG is delta 0.5 range 0.0 .. 24.0;

  type TIDSTABEL is array (DAG) of TIMEFORBRUG;

  ARBEJDSTID: constant TIDSTABEL:=(MANDAG..FREDAG=>8.0,
                                     LØRDAG|SØNDAG =>0.0);

end ARBEJDS_DATA;
```

Denne pakke samler logisk sammenhængende datatyper i ét modul.

Modulær programmering

Oftest vil man dog samle såvel datatyper som objekter og operationer herpå. Sådanne moduler kaldes undertiden abstrakte datastrukturer, da de kan benyttes til at udvide sproget med mere abstrakte typer og operationer end de indbyggede. Som eksempel herpå gengives reference-manualens eksempel på den abstrakte datatype rationale tal. Her indeholder pakken såvel en specifikationsdel som en krop:

```

package RATIONALE_TAL is

  type RATIONAL is
    record
      TÆLLER: INTEGER;
      NÆVNER: INTEGER range 1..INTEGER'LAST;
    end record;

  NUL: constant RATIONAL := (0,1);

  function ENS(X, Y: RATIONAL) return BOOLEAN;

  function "+"(X, Y: RATIONAL) return RATIONAL;

end;

package body RATIONALE_TAL is

  procedure SAMME_NÆVNER(X, Y: in out RATIONAL) is
    begin
      -- forlænger X og Y, så de får samme nævner
    end;

  function ENS(X, Y: RATIONAL) return BOOLEAN is
    U, V: RATIONAL := NUL;
  begin
    U := X; V := Y;
    SAMME_NÆVNER(U,V);
    return U.TÆLLER = V.TÆLLER;
  end ENS;

  function "+" (X, Y: RATIONAL) return RATIONAL is
    ...
  end "+";

end RATIONALE_TAL;

```

Eksemplet illustrerer også, hvorledes der i kroppen kan erklæres og benyttes funktioner (her SAMME_NÆVNER), som brugeren af pakken ikke har adgang til. Funktionerne kan ikke "ses" udefra, da de ikke er nævnt i specifikationsdelen.

Modulær programmering

Som det fremgår af eksemplet kan man benytte operator-symboler (her "+") som navn for funktioner. Herved udvides definitionen af den pågældende operator til at dække den nye "abstrakte" datatype.

Et program, der anvender denne pakke, kunne tage følgende form:

```
procedure HOVEDPROGRAM is

  package RATIONALE_TAL is
    .
    .
    .
  end;

  use RATIONALE_TAL;      -- gør indmaden i pakken
                          -- synlig i resten af
                          -- proceduren

  A, B, C: RATIONAL := NUL;

  package body RATIONALE_TAL is
    .
    .
    .
  end RATIONALE_TAL;

begin

  if ENS(A,B) then
    B := C + (7,9) + (1,1);
  end if;

end;
```

use-sætningen gør de navne, som er erklærede i pakkens specifikationsdel, synlige i den efterfølgende del af procedure HOVEDPROGRAM. Man kan således referere direkte til funktionen ENS uden at bruge det fulde navn, som ville være RATIONALE_TAL.ENS.

Modulær programmering

Brugeren af pakken kan indføre nye operationer på rationale tal, f. eks. kunne han erklære følgende funktion lige efter use ovenfor:

```
function SJOV(X: RATIONAL) return RATIONAL is
    Y: RATIONAL := NUL; -- hjælpevariabel
begin
    Y.TELLER := X.TELLER - 1;
    Y.NEVNER := X.NEVNER;
    return Y;
end;
```

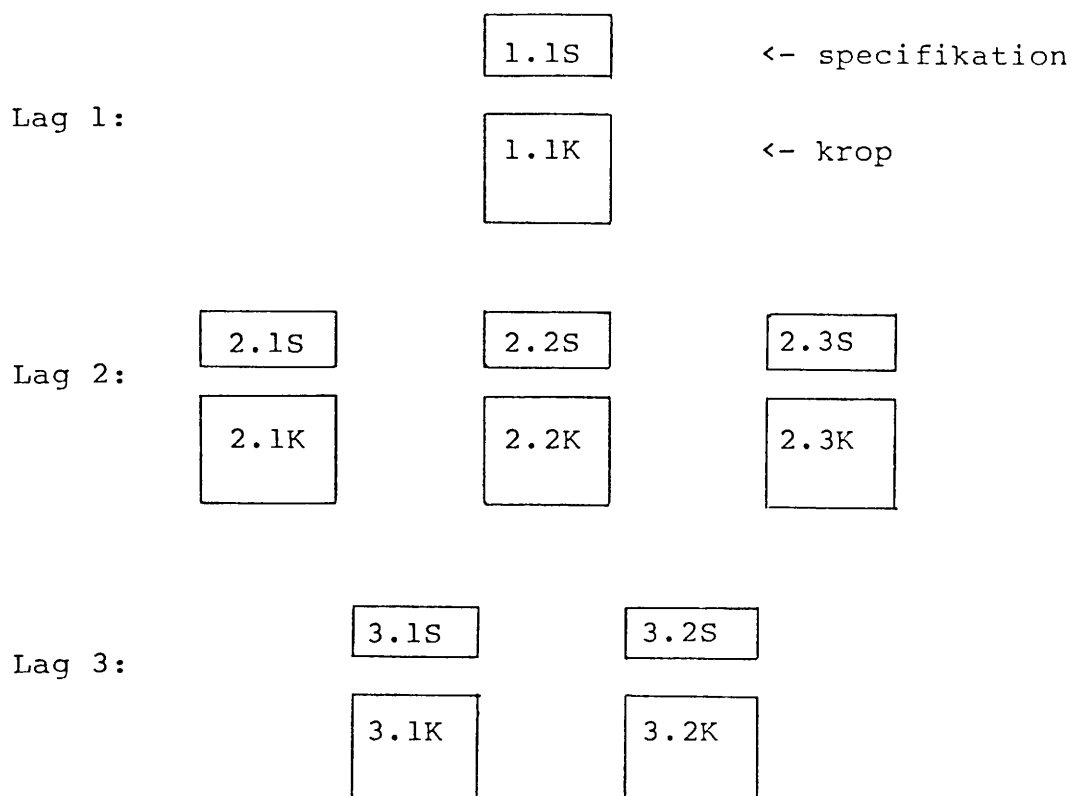
Her udnytter brugeren sin viden om, hvordan typen RATIONAL er opbygget, og han har skabt en operation, der normalt er meningsløs i forbindelse med rationale tal. Det er muligt at omformulere pakken, så den realiserer en indkapslet datatype ("encapsulated data type"), dvs. en type hvorpå kun pakkens egne operationer kan udføres. Specifikationsdelen af en sådan pakke vises her:

```
package RATIONALE_TAL is
    type RATIONAL is private;
    NUL: constant RATIONAL;
    function ENS(X, Y: RATIONAL) return BOOLEAN;
    function "+"(X, Y: RATIONAL) return RATIONAL;
private -- informationen nedenfor er ikke synlig udefra
    type RATIONAL is
        record
            TELLER: INTEGER;
            NEVNER: INTEGER range 1..INTEGER'LAST;
        end record;
    NUL: constant RATIONAL := (0,1);
end;
```

Kroppen har samme udseende som i eksemplet side 6-3.

Modulær programmering

Et Ada program kan opbygges af hierarkier eller lag af programmoduler, f. eks.:



En regel kunne være: Et modul på lag 'i' kan benytte moduler på lag 'i' og 'i+1'.

Programmet kan programmeres 'bottom up' ved at begynde med det nederste lag, hvor både specifikationsdel og krop kodes og testes; dernæst går man videre til det næste lag osv.:

3.1S, 3.2S, 3.1K, 3.2K, 2.1S, 2.2S, ...

eller 'top down':

1.1S, 1.1K, 2.1S, 2.2S, 2.3S, 2.1K, 2.2K, ...

En iteration mellem udviklingen af komponenterne vil normalt være nødvendig.

Desuden er diverse blandinger mulige, f. eks. fastlæggelse af alle specifikationer først, dernæst kodning af kropene.

Takket være muligheden for separat oversættelse kan flere grupper arbejde parallelt med hver sit modul, og alligevel få afprøvet grænsefladerne på et tidligt tidspunkt.

Modulær programmering

6.2 Separat oversættelse

Et Ada program er en samling af ét eller flere oversættelsesmoduler, som er blevet oversat i én eller flere omgange. Et oversættelsesmodul kan være en pakke, et underprogram eller en stub (se nedenfor). Også generiske pakker og underprogrammer kan oversættes separat (generiske moduler beskrives senere). Ada oversætteren holder rede på de afhængigheder, der findes mellem de oversatte moduler. Hvis et modul A afhænger af B (f. eks. fordi A kalder et underprogram i B), skal B's specifikation oversættes før kroppen af A. Hvis B's specifikation senere ændres og oversættes igen, må Ada oversætteren enten af sig selv genoversætte A, eller mærke A som værende inkompatibel med B samt meddele brugeren dette.

Tænker vi os, at pakken med rationale tal er blevet oversat separat og derved blevet inkluderet i et program-bibliotek, bliver den tilgængelig for anvendelse i andre moduler via with sætningen. with sætningen angiver de moduler, som ønskes gjort tilgængelige. use sætningen kan dernæst benyttes som tidligere nævnt.

```
with RATIONALE_TAL;
```

```
procedure HOVEDPROGRAM is
```

```
    use RATIONALE_TAL;
```

```
    A, B, C: RATIONAL := NUL;
```

```
    begin
```

```
        ...; -- som i det tidligere eksempel
```

```
    end;
```

Modulær programmering

Det er muligt at oversætte dele af et oversættelsesmodul separat uden at denne del, der kaldes en stub, bliver et biblioteksmodul, og dermed tilgængelig for andre:

```
procedure HOVEDPROGRAM is
  package RATIONALE_TAL is
    .
    .               -- specifikationsdel
    .
  end;

  use RATIONALE_TAL;

  A, B, C: RATIONAL := NUL;

  package body RATIONALE_TAL is separate;

begin
  .
  .
  .
end;
```

Proceduren HOVEDPROGRAM oversættes før kroppen af RATIONALE_TAL ('top down'). Ved oversættelse af kroppen må det miljø, kroppen tilhører, angives:

```
separate(HOVEDPROGRAM)

package body RATIONALE_TAL is
  .
  .
  .
end RATIONALE_TAL;
```

Også undermoduler af undermoduler (etc.) kan oversættes separat - f. eks. funktionen ENS fra RATIONALE_TAL:

```
separate(HOVEDPROGRAM.RATIONALE_TAL)

function ENS(X, Y: RATIONAL) return BOOLEAN is
  .
  .
  .
end;
```

Modulær programmering

6.3 Generiske moduler

Underprogrammer og pakker kan være generiske. Generiske programmoduler er skabeloner af programmoduler og vil ofte være parameteriserede. Da de er skabeloner, kan de ikke benyttes direkte som almindelige pakker og underprogrammer. Man kan f. eks. ikke kalde en generisk funktion. Ud fra disse skabeloner kan man imidlertid skabe (generere) forekomster; sådanne forekomster har samme egenskaber som almindelige pakker og underprogrammer og kan benyttes direkte. En forekomst vil under skabelsen blive 'præget' med de aktuelle parametre svarende til den generiske skabelons formelle parametre. Man overfører således parametre til skabelonen, når man genererer en forekomst. Disse parametre kan bl. a. være typer og underprogrammer.

Her skal vises et simpelt eksempel, en skabelon for en procedure, der ombytter indholdet i to variable af vilkårlig type:

```
generic

  type UKENDT is private;

  procedure OMBYT_GENERATOR(X, Y: in out UKENDT) is

    Z: UKENDT; -- hjælpevariabel af den (endnu)
                -- ukendte type

  begin
    Z := X; X := Y; Y := Z;
  end;
```

Antager vi, at denne generiske procedure er blevet oversat separat, og at dette også gælder pakken RATIONALE_TAL i den form, der er vist side 6-3, kan et program generere en forekomst af denne procedure til ombytning af rationale tal:

```
with RATIONALE_TAL, OMBYT_GENERATOR;

procedure HOVEDPROGRAM is

  use RATIONALE_TAL;
  A, B: RATIONAL := NUL;
  procedure OMBYT is new OMBYT_GENERATOR(RATIONAL);

  begin
    A := (1,7); B := (8,8);
    OMBYT(A,B);
    -- nu er A lig med (8,8), og B lig med (1,7)
  end;
```

Modulær programmering

Hvis vi har brug for at kunne ombytte heltal, kan vi desuden generere en "heltals-OMBYT":

```

with RATIONALE_TAL, OMBYT_GENERATOR;

procedure HOVEDPROGRAM is

    use RATIONALE_TAL;
    A, B: RATIONAL := NUL;
    I, J: INTEGER := 0;

    procedure OMBYT is new OMBYT_GENERATOR(RATIONAL);
    procedure OMBYT is new OMBYT_GENERATOR(INTEGER);

begin

    A := (1,7);
    B := (8,8);
    OMBYT(A,B);
    -- nu er A lig med (8,8), og B lig med (1,7)

    I := 17;
    J := -9987;
    OMBYT(I,J);
    -- nu er I = -9987 og J = 17

end;

```

Da typerne INTEGER og RATIONAL er forskellige, kan oversætteren af sammenhængen se, hvilken OMBYT der skal kaldes. Denne form for navnesammenfald kaldes "overloading".

Ved hjælp af sådanne generiske moduler kan man én gang for alle opbygge programbiblioteker, der opererer på hyppigt forekommende abstrakte datastrukturer, såsom stakke og køer. Tilpasningen til en specifik datatype sker først, når en forekomst skal genereres. Dette kan blandt meget andet bruges til at konstruere generelt anvendelige I/O procedurer.

Grænseflader til omgivelser

7. Grænseflader til omgivelser

I dette indlæg behandles nogle af de faciliteter i Ada, som giver et program mulighed for at kommunikere med sin omverden. Det drejer sig om :

- Inddata og uddata.
- Kald af underprogrammer skrevet i andre programmeringssprog.
- Brug af underprogrammer skrevet i maskinkode.
- Intern repræsentation af data.
- Styring af ydre enheder, herunder behandling af afbrydelser ("interrupts").

7.1 Inddata/uddata

I modsætning til de fleste andre højniveau programmeringssprog har man i Ada defineret underprogrammer for behandling af inddata og uddata som en del af det standardiserede sprog. Disse underprogrammer er definerede i særlige program-pakker. Dette skyldes ønskerne om flytbarhed af Ada programmer. Man vil forhindre, at hver oversætter har sin specielle definition af inddata og uddata faciliteter.

Alle i/o operationer i Ada foregår på eksterne filer. En ekstern fil kan være hvadsomhelst, når blot den er i stand til at producere en værdi, der kan indlæses, eller modtage en værdi, som udlæses fra programmet. En ekstern fil identificeres ved en tegnstreng (filnavnet). Det fysiske objekt, som svarer til en Ada fil kan f. eks. være en pladelager-fil, en magnetbånds-fil, en linieskriver, en papirbåndslæser, en relæudgang eller en skærmterminal.

Eksterne filer kan være organiseret med sekventiel eller direkte tilgang. Disse tilgangsformer er nærmere forklaret i det følgende.

I resten af dette kapitel bruges ordet "fil" synonymt med "ekstern fil".

Grænseflader til omgivelser

Der er defineret følgende pakker til brug for operationer på filer:

SEQUENTIAL_IO	definerer et antal i/o operationer til ind- og udlæsning af data med sekventiel tilgang,
DIRECT_IO	definerer et antal i/o operationer til ind- og udlæsning af data med direkte tilgang,
TEXT_IO	definerer et antal i/o operationer til ind- og udlæsning på en tegnform,
IO_EXCEPTIONS	definerer standard fejlbehandling for i/o fejl opdaget i pakkerne SEQUENTIAL_IO, DIRECT_IO og TEXT_IO,
LOW_LEVEL_IO	definerer visse lav-niveau I/O operationer til direkte styring af ydre enheder.

Ved sekventiel tilgang overføres data i den rækkefølge hvori de præsenteres for i/o systemet.

Ved direkte tilgang opfattes filen som bestående af et antal elementer, der optager nummererede positioner i lineær orden. En dataværdi kan overføres til eller fra et vilkårligt element i filen. Elementet, hvortil eller hvorfra en overførsel skal ske udpeges ved et indeks, som er et positivt heltal.

Grænseflader til omgivelser

7.2 SEQUENTIAL_IO og DIRECT_IO pakkerne

SEQUENTIAL_IO og DIRECT_IO er generiske pakker, dvs. pakkeskabeloner, som inden de kan bruges i et program skal parametriseres med den ønskede elementtype.

Eksempel:

Vi ønsker at generere en sekventiel uddatafil der udelukkende indeholder heltal. Først parametriseres SEQUENTIAL_IO:

```
package HELTAL_IO
  is new SEQUENTIAL_IO( ELEMENT_TYPE => INTEGER );
use HELTAL_IO;
```

Derefter oprettes en fil :

```
RESULTATER: FILE_TYPE;
CREATE( FILE => RESULTATER,
        MODE => OUT_FILE,
        NAME => "USERS.JJ.RES" );
```

Filen er nu åben for skrivning :

```
WRITE( FILE => RESULTATER, ITEM => 1981 );
WRITE( RESULTATER, 25 );
WRITE( RESULTATER, X );
```

Når skrivning er tilendebragt lukkes filen :

```
CLOSE( RESULTATER );
```

På et senere tidspunkt ønsker vi at læse filen. Først åbnes filen:

```
GAMLE_RES: FILE_TYPE;
OPEN( GAMLE_RES, IN_FILE, "USERS.JJ.RES" );
```

Derefter kan posterne læses:

```
READ( GAMLE_RES, X );
```

Når læsning er tilendebragt kan filen lukkes eller slettes. Lad os antage, at vi ønsker at slette filen:

```
DELETE( GAMLE_RES );          -- filen skal være åben
```

Grænseflader til omgivelser

I det følgende gives en oversigt over de vigtigste procedurer og funktioner som findes både i `SEQUENTIAL_IO` og i `DIRECT_IO`. For overskuelighedens skyld er underprogramdefinitionerne simplificeret en smule. De fuldstændige og korrekte definitioner findes i referencemanualen, afsnit 14.2.

```
procedure CREATE( FILE : in out FILE_TYPE;
                  MODE : in      FILE_MODE;
                  NAME : in      STRING;
                  FORM : in      STRING      );
```

opretter en fysisk fil og tilknytter denne til Ada-filen FILE.

```
procedure OPEN(   FILE : in out FILE_TYPE;
                  MODE : in      FILE_MODE;
                  NAME : in      STRING;
                  FORM : in      STRING      );
```

tilknytter en eksisterende fysisk fil til Ada-filen.

```
procedure CLOSE(  FILE : in out FILE_TYPE );
```

bryder forbindelsen mellem den fysiske fil og Ada-filen.

```
procedure DELETE( FILE : in out FILE_TYPE );
```

fjerner den pågældende fysiske fil.

```
procedure RESET(  FILE : in out FILE_TYPE;
                  MODE : in      FILE_MODE  );
```

Spoler filen tilbage så næste læsning fra eller skrivning til Ada-filen sker fra det første element.

```
function NAME( FILE : in FILE_TYPE ) return STRING;
```

returnerer det eksterne navn der svarer til den udpegede Ada-fil.

```
function IS_OPEN( FILE : in FILE_TYPE ) return BOOLEAN;
```

giver svaret TRUE hvis der eksisterer en forbindelse mellem Ada-filen og en fysisk fil, dvs. hvis filen er åben.

Grænseflader til omgivelser

Herudover findes en række underprogrammer som er specifikke for SEQUENTIAL_IO hhv. DIRECT_IO. Definitionerne af de vigtigste af disse underprogrammer i pakken. DIRECT_IO er vist nedenfor. De tilsvarende definitioner i SEQUENTIAL_IO er ikke vist men minder kraftigt om de viste.

```
procedure READ(  FILE : in    FILE_TYPE;
                  ITEM  : out    ELEMENT_TYPE;
                  FROM  : in    POSITIVE_COUNT );
```

indlæser dataværdien fra elementet udpeget af indeks anført i FROM i den eksterne fil til ITEM. Indeks forøges med 1.

```
procedure WRITE( FILE : in    FILE_TYPE;
                  ITEM  : in    ELEMENT_TYPE;
                  TO    : in    POSITIVE_COUNT );
```

udlæser dataværdien i ITEM til elementet udpeget af indeks anført i TO til den eksterne fil. Indeks forøges med 1. Ændrer evt. også slutpositionen og filstørrelsen.

```
function SIZE(  FILE : in    FILE_TYPE ) return COUNT;
```

returnerer antallet af elementer der på kaldetidspunktet befinder sig i den eksterne fil.

```
function END_OF_FILE(FILE : in FILE_TYPE) return BOOLEAN;
```

returnerer TRUE hvis den nuværende indekxsværdi overstiger antallet af elementer i filen - ellers returneres FALSE.

Grænseflader til omgivelser

7.3 TEXT_IO pakken

Vel nok den mest almindelige form for inddata og uddata udgøres af tekster, som er læsbare for mennesker. Derfor er der specielt til dette brug defineret en pakke, som har en række udvidede operationer i forhold til SEQUENTIAL_IO og DIRECT_IO.

Elementtypen for en tekst-fil er CHARACTER. En tekstfil betragtes som værende opdelt i sider, som er opdelt i linier, som så igen er opdelt i et antal tegn. Siderne, linierne og tegnene er nummereret startende med nr. 1. Tegnene i en linie siges at have et søjlenummer, første søjle har nummeret 1.

En åben fil har tilknyttet 3 størrelser :

- Aktuelt søjle nummer.
- Aktuelt linie nummer.
- Aktuelt side nummer.

Ud over READ og WRITE, som kendes fra SEQUENTIAL_IO og DIRECT_IO er der i TEXT_IO defineret to procedurer til læsning hhv. skrivning, nemlig GET og PUT. Disse procedurer foretager de nødvendige konverteringer mellem intern repræsentation og ASCII tegnstreng, f. eks. konvertering af et binært heltal fra/til en decimal repræsentation bestående af tegnene "0" til "9".

I TEXT_IO er der defineret yderligere 4 pakker til brug for numeriske typer og opremsnings ("enumeration") typer

- INTEGER_IO
- FLOAT_IO
- FIXED_IO
- ENUMERATION_IO.

Disse er generiske og må altså parametriseres med den ønskede type.

ENUMERATION_IO giver mulighed for udskrivning og indlæsning af tegnstreng, som repræsenterer værdier for variable af opremsningstyper.

TEXT_IO giver således mulighed for nemt at konstruere pakkør til specielle formål. Hvis man f.eks. har brug for at udskrive en tabel med flere søjler (af forskellig art), kan man parametrisere TEXT_IO med en record-type, som indeholder felter til beskrivelse af en udskrifts-linie. TEXT_IO er jo også et eksempel på brug af SEQUENTIAL_IO eller DIRECT_IO med element-type CHARACTER.

Grænseflader til omgivelser

Der er i TEXT_IO defineret en lang række procedurer og funktioner ikke blot til læsning og skrivning men også til manipulation af søjlenumre, linienumre og linielængder. For detaljer henvises til Ada reference-manualen.

Det er muligt at bruge I/O operationer fra TEXT_IO uden selv at specificere nogen filer. To systemdefinerede filer vil blive brugt i dette tilfælde (STANDARD_INPUT og STANDARD_OUTPUT).

Et eksempel på brugen af TEXT_IO er vist i afsnit 4.5.

7.4 LOW LEVEL_IO pakken

Denne pakke benyttes til kontrol af ydre enheder.

Proceduren SEND_CONTROL bruges til at sende kontrol information til en ydre enhed.

Proceduren RECEIVE_CONTROL bruges til at modtage status/kontrol information.

Da formater for kontrol information varierer betydeligt med den aktuelle ydre enhed, vil typerne for de formelle parametre til disse procedurer være implementationsbestemte.

Grænseflader til omgivelser

7.5 Grænseflader til andre programmeringssprog

Det er muligt i Ada programmer at kalde underprogrammer skrevet i andre programmeringssprog. At et underprogram er skrevet i et andet sprog indikeres v.h.a. et såkaldt pragma INTERFACE :

```
package FORTRAN_BIBLIOTEK is
  function SQRT(X : FLOAT) return FLOAT;
  function EXP(X : FLOAT, Y : INTEGER) return FLOAT;
private
  pragma INTERFACE(FORTRAN, SQRT);
  pragma INTERFACE(FORTRAN, EXP);
end FORTRAN_BIBLIOTEK;
```

Det vil afhænge af implementationen, hvilke restriktioner der lægges på kald af underprogrammer skrevet i andre sprog.

En procedurekrop kan indeholde maskinkode, hvis den har en PRAGMA INLINE knyttet til sig og v.h.a. use anvender en pakke, hvori denne maskinkode er defineret på symbolsk form. Overalt, hvor en sådan procedure kaldes, vil der i objekt-koden blive lagt en kopi af maskinkoden. Der vil altså ikke være tale om et procedurekald i egentlig forstand.

```
procedure P; pragma INLINE(P);
procedure P is
  use INSTRUKTIONER_CR80D;
begin
  MOVE'(FROM => R0, TO => R7);
  .
  .
  .
end P;
```

Grænseflader til omgivelser

7.6 Grænseflader til den underliggende maskine7.6.1 Repræsentationsspecifikationer

Det vil normalt være oversætteren, der bestemmer den interne repræsentation af data. Det er imidlertid muligt selv at styre dette. Nogle få eksempler skal vises her:

-- størrelse af objekt :

```
type T is delta 0.01 range -100 .. 100;
```

```
for T'SIZE use 15;  -- 15 bit til repræsentation af
                    -- objekter af denne type.
```

-- data areal til hver aktivering af en given proces:

```
task type T is
```

```
  .
```

```
  .
```

```
end;
```

```
SIDE : constant := 1024;
```

```
for T'STORAGE_SIZE use 4.SIDE;
```

-- positionsværdier for opremsningstyper :

```
type FARVE is (RØD, GRØN, GUL, BRUN);
```

```
for FARVE use (RØD=>1, GRØN=>2, GUL=>4, BRUN=>8);
```

-- Adresse specifikationer :

```
X : INTEGER;
```

```
for X use at 2048;  -- placer X i ord 2048 i lageret
```

7.6.2 Afbrydelser

Det kan for et ENTRY i et givet TASK specificeres at dette skal kaldes i tilfælde af en afbrydelse fra en given I/O-adresse :

```
task INTERRUPT_HANDLER is
```

```
  entry ORDRE_UDFØRT;
```

```
  for ORDRE_UDFØRT use at 32;
```

```
end;
```

Ada lærebøger

8. Ada lærebøger

Der er gennem de sidste år fremkommet en sand syndflod af bøger og artikler om Ada. Den følgende liste giver en oversigt over de bøger og artikler om Ada, som DDC har anskaffet til sit bibliotek. Nogle af disse bøger er efter vores mening velegnede som introduktion til sproget:

J.G.P. Barnes

An Overview of Ada

Software - Practice and Experience, November 1980,
Vol. 10, No. 11, page 851-887.

Udmærket kortfattet oversigt på artikelform over de væsentligste nyskabelser i Ada. Denne artikel omhandler juli 1980 udgaven af Ada.

J.G.P. Barnes

Programming in Ada

Addison-Wesley Publishing Co. 1982, 340 sider.

Denne bog var efter vores opfattelse den bedste på markedet ind til Grady Booch udsendte sin bog (se nedenfor). Den behandler hele Ada sproget rimelig grundigt og er skrevet med stor indsigt af en af nøglepersonerne fra Ada udviklingsholdet. Bogen er skrevet i en tør humoristisk stil, der er ganske underholdende.

Grady Booch

Software Engineering With Ada

The Benjamin/Cummings Publishing Co. 1983, ca. 530 sider.

Den eneste Ada lærebog, som vi kender, der ikke bare forsøger at bibringe læseren en forståelse af Adas syntaks og semantik, men som også beskæftiger sig med sunde principper for programmelkonstruktion. Bogen illustrerer med velgennemarbejdede eksempler, hvorledes Ada sproget effektivt understøtter brugen af disse principper.

Bogen er med rette den mest anvendte grundbog for egentlige Ada kurser. DDC anvender denne bog som grundbog for sine Ada kurser.

Man kan købe færdige plancher til bogen. Den typografiske kvalitet af disse plancher er dog ikke imponerende.

Ada lærebøger

Valerie A. Downes and Stephen J. Goldsack
Programming Embedded Systems With Ada
Prentice Hall International 1982, ca. 390 sider

Behandler anvendelsen af Ada i indkapslede systemer ("embedded systems") ved at beskrive programudviklingen af et datamatiseret patientovervågningssystem i Ada.

Narain Gehani
Ada - An Advanced Introduction
Prentice Hall 1983, ca. 345 sider.

Henvender sig særlig til programmører med et forudgående kendskab til Pascal, C, PL/1, Algol 60 eller Fortran.

Peter Hibbard m.fl.
Studies in Ada Style
Springer Verlag 1981, 104 sider.

Indeholder en række interessante og velgennemarbejdede ikke-trivielle eksempler på anvendelser af de avancerede konstruktioner i Ada, fx drivprogram til PDP-11 og løsning af differentialligningssystem ved hjælp af samarbejdende parallelle processer. Der er i alt 5 eksempler på 100 - 300 linier hver. Bogen er ikke egnet som første lærebog i Ada.

Brian Mayoh
ADA - en introduktion
teknisk forlag a-s 1982, 40 sider.

Foreløbig den eneste bog på dansk om Ada. Meget kortfattet.

Brian Mayoh
Problem Solving With ADA
John Wiley & Sons 1982, 233 sider.

Denne bog er en lærebog i problemløsning. Den er ikke velegnet som introduktion til Ada. Det forventes at denne bog udkommer i en dansk udgave i april 1984 fra teknisk forlag.

Ada lærebøger

I.C. Pyle

The ADA Programming Language

Prentice Hall International 1981, 294 sider.

En af de første lærebøger om Ada. Omhandler juli 1980 udgaven af Ada. Bogen har en kedelig typografi.

Peter Wegner

A Self-Assessment Procedure Dealing With the
Programming Language Ada

Communications of the ACM, October 1981, Vol. 24,
No. 10, page 647-677.

37 omhyggeligt gennemarbejdede spørgsmål om alle aspekter af Ada. Hvert spørgsmål er forsynet med en grundig introduktion, der betyder at forudgående kendskab til Ada ikke er nødvendigt. Ved hjælp af denne artikel kan man kontrollere og udbygge sin viden om de vigtigste faciliteter i Ada.

Peter Wegner

Programming with Ada, An Introduction by Graduated
Examples

Prentice Hall International, second edition 1982.

En introduktion til Ada beregnet på den erfarne programmer. Beskrivelsen af Ada sker udelukkende ved eksempler.

Ada referencemanualen er efter vores opfattelse ikke velegnet som lærebog i Ada.