Kapitel 4:
Kunsten at få en computer til at adlyde (I): Programmering

Henrik Kragh Sørensen Mikkel Willum Johansen

22. april 2022

Indhold

4.1 Modularisering, abstraktion og grænseflader 1
4.2 Programmeringssprog 3
4.3 Programmeringsparadigmer 11
4.4 Fra algoritme til afvikling 13
4.5 Fuld fart frem 16

Litteratur **18**

4.1 Modularisering, abstraktion og grænseflader

På det tekniske niveau består en afvikling af et program på en moderne computer principielt i, at processoren henter instruktionerne en ad gangen, afkoder instruktionen og udfører den. Betragtet på den vis, er et computerprogram en streng af instruktionskoder — det, vi i kapitel 3 har kaldt *software* eller *maskinkode*. I nogle sammenhænge, hvor plads eller hastighed er voldsomt begrænsende faktorer, kan det være nødvendigt for ingeniøren at programmere computeren direkte i maskinkode, kun hjulpet af en såkaldt *assembler*, der ikke stiller ret megen understøttelse til rådighed for programmøren. Men i de fleste andre tilfælde foregår programmeringen af den fysiske computer igennem en lang række lag af abstraktioner, modulariseringer og oversættelser. I dette afsnit skal vi først betragte abstraktioner og modulariseringer, inden vi i afsnit 4.2 fokuserer på programmeringssprog.

Når man skal designe software er der mange hensyn at afbalancere, og vægtningen kan variere mellem forskellige projekter: Nogle gange er pålidelighed afgørende, nogle gange er hastighed vigtigst, nogle gange skal man prioritere omkostningerne ved at vedligeholde koden over en lang årrække. Og som regel spiller alle disse hensyn nært sammen uden at programmøren har en eksplicit *videnskabelig* teori at basere sine valg på. Derfor er softwareudvikling ofte blevet betegnet som en mere ingeniøragtig praksis, hvor erfaringer og 'best-practices' spiller en stor rolle, eller endda som en artistisk udfoldelse, sådan som Donald Knuth har argumenteret for (se Bond, 2005).

Men også softwareudvikling — og især under termen *software engineering* — har stræbt efter at finde et kollektivt grundlag at bygge på i form af noget, der kunne ligne et *paradigme* (Northover m.fl., 2008). Selve begrebet *software engineering* blev første gang brugt under en meget omtalt og betydningsfuld konference arrangeret af NATO i Garmisch i Vesttyskland

i 1968 (se fx MacKenzie, 2001). Konferencen var blevet indkaldt, fordi man i løbet af 1960'erne havde oplevet mange software-projekter gå over tid og over budget eller ikke leve op til forventningerne: Man stod i en *softwarekrise*. Udvikling af software havde, med andre ord, vist sig svær at kontrollere, og det havde stor bevågenhed både fra militær side og fra de store civile producenter. Og vejen ud af krisen syntes at være at gøre udviklingen af software til en *ingeniørvidenskab*, som kunne bygge bro mellem den akademiske, teoretiske viden og praktikerens behov. Men fornemmelsen af krise i 1960'erne førte også til, at man søgte at udvikle og forfine processer og især programmeringssprog, der kunne understøtte effektiv og pålidelig software.

Blandt de mange nye ideer, som *softwarekrisen* førte med sig, var ideen om *struktureret programmering*, som særligt den hollandske datalog Edsger Dijkstra (1930–2002) havde fremført fra 1966 og som han forsvarede indgående og polemisk i et opgør med den ellers så anvendte `goto`-kommando (Dijkstra, 1968). Alle instruktionssæt havde en primitiv kommando `JUMP`, der flyttede programtælleren til en given adresse i hukommelsen, og denne egenskab var naturligt også blevet medtaget i de tidligere programmeringssprog, herunder også det populære sprog ALGOL (se Rutishauser, 1967).

> The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. (Dijkstra, 1968, s. 147)

Dijkstras indvendinger gik blandt andet på, at *udførelsen* af et program, der indeholder mange `goto`-kommandoer, blev uoverskuelig, og det betød blandt andet, at det blev meget sværere at ræsonnere omkring programmet, herunder at finde fejl i det.

Alternativet blev en ny generation af *strukturelle sprog*, der understøttede Dijkstras ide om *struktureret programmering* ved at bryde kildekoden op i strukturelle portioner med subroutiner, blokke og kontrolstrukturer, hvorved det blev muligt bedre at bevare overblikket over kodens udførelse.

En anden væsentlig indsigt, som kom ud af *softwarekrisen* i 1960'erne, var gevinsten ved *modularisering*: Ved at bryde store systemer op i mindre og selvindeholdte dele, kunne man dels genbruge samme løsning i forskellige delproblemer og dels arbejde på at gøre de enkelte moduler så robuste og fejlfrie som muligt. Denne tænkning havde sådan set været med helt fra computerens fødsel, hvor nogle af de første programmer, der blev udviklet, var moduler, der kunne udregne matematiske funktioner og konstanter, og som kunne bruges ved indsættelse bestemte steder i koden. Oprindeligt betød indsættelse her, at man rent fysisk skiftede båndet med maskinkoden ud, men senere blev det en væsentlig del af den proces, vi stadig kalder *linking*, hvor det nye program forbindes til de biblioteker af allerede udviklede redskaber, som i dag er en stor del af programmeringssprogene.

Samlet set kan vi betragte både struktureret programmering og modularisering som eksempler på en abstraktionsproces, hvor de mere hardware-nære elementer af programmeringen skjules for programmøren igennem en *grænseflade*. Og den ide — om lag af abstraktioner adskilt af ekspliciterede grænseflader — er en af de mest gennemgående ideer i datalogien. Vi har allerede set den i forbindelse med formel verifikation af software, og vi skal møde den igen i opfattelsen af den stak af oversættelser, der omdanner en algoritme til en fysisk eksekvering (se afsnit 4.4). Men også i tænkningen om programmeringssprog kan denne ide om abstraktion og grænseflade vise sig nyttig, hvilket vi skal behandle i næste afsnit.

Særligt vigtigt er det, at abtraktion væk fra hardware i retning mod programmørens behov og begrebsverden har gjort det muligt at tænke om programmeringssprog og biblioteker (API'er) som *abstrakte computere*, der kan tilgås og programmeres ved hjælp af et sæt

af højereordens begreber (interfaces, grænseflader). Og nogle af disse *abstrakte* computere bliver faktisk instantieret i form af *virtuelle maskiner*, hvor en (måske fysisk) computer *emulerer* udførslen af en anden, *virtuel* maskine. Mest udbredt er dette sikkert i form af virtuelle maskiner, der tillader at emulere et operativsystem som et vindue under et andet operativsystem, men den samme teknologi bliver brugt i mange kritiske systemer til at *indeslutte* (encapsulate) potentielt usikre elementer, så de ikke kan komme til at påvirke de kritiske funktioner (se fx arkitekturen beskrevet i Klein m.fl., 2018).

4.2 Programmeringssprog

Den østrigske datalog Niklaus Wirth er en af de store specialister i design af programmeringssprog, og han beskriver den rolle, som programmeringssprog bør spille således:

> A language represents an abstract computer whose objects and constructs lie closer to, and reflect more directly, the problem to be represented than the concrete machine. (Wirth, 2008, s. 34)

Denne måde at betragte et sprog som en abstrakt computer, der ligger nærmere programmørens formål end den fysiske maskine, er et meget velvalgt perspektiv, som Wirth kan anlægge retrospektivt. Men i hele datalogiens historie har problemet med på effektiv vis at give computeren instruktioner til udførelse været en stor udfordring.

Sprog som abstraktioner

I slutningen af 1950'erne — inden for bare få år — blev denne udfordring mødt på fire ganske forskellige måder, betinget af forskellige hensyn (se også figur 2):

1. International Business Machines (IBM) var den markedsdominerende producent af hardware, og dengang betragtede man software som en del af den hardware, man betalte penge for se også Haigh, 2002: En IBM computer var med andre ord ikke så meget værd, hvis der fandtes brugbar software til den, og opgaven med at producere denne software tilfaldt hardwareproducenten, *in casu* IBM (Haigh, 2002). Derfor udviklede IBM i 1957, under ledelse af John Backus (1924–2007), programmeringssproget FORTRAN til især deres kunder ved forskningsinstitutioner, hvor der var stort behov for numeriske beregninger og simulationer. FORTRAN stod for 'FORmula TRANslation', og sproget var udviklet med dette formål for øje: At gøre det 'nemt' for forskere og andre at programmere deres beregninger og sikre, at de blev udført korrekt og hurtigt.

2. Som et delvist modsvar til IBM's dominans samledes en gruppe akademikere og andre sig om at beskrive det sprog, der i 1958 skulle blive beskrevet som ALGOL (ALGOrithmic Language), og med ALGOL-60 skulle blive en stor og varig succes. Gruppen bag ALGOL ønskede, at sproget skulle udgøre et akademisk og europæisk alternativ til IBM's FORTRAN, hvilket blandt andet havde betydning for, at ALGOL var *maskinagnostisk* og blev oversat til mange forskellige arkitekturer, herunder de danske computere DASK og GIER (Vilstrup, 1963). Udviklingen af ALGOL var et fælles anliggende (drevet af *CUDOS*-lignende normer), men blandt de ledende drivkræfter var amerikanerne Backus og Alan Perlis (1922–1990) og — særligt — danske Peter Naur (1928–2016), der fik det koordinerende hovedkvarter (og nyhedsbrevet)

flyttet til København (Nofre, 2010). Selve sproget var designet til tunge akademiske beregninger, men også som en forsøgsplads for algoritmiske eksperimenter.

3. Allerede det følgende år, i 1959, blev et tredje vigtigt programmeringssprog klar, som satte en helt ny retning på programmeringssprogenes formål. Ved det prestigiøse Stanford University i Californien var forskningen i kunstig intelligens (AI) inde i en rivende udvikling (se også kapitel 6), og John McCarthy (1927–2011) var sammen med Marvin Minsky (1927–2016) en af de forskere, der spændte over både datalogi og kognitionsvidenskab. Til brug for forskningen i AI udviklede McCarthy programmeringssproget LISP, som var det første eksempel på et *funktionelt programmeringssprog*, hvor abstraktionen fra den underliggende hardware var gennemført for at stille en abstrakt maskinmodel til rådighed for forskere, der var interesserede i at modellere menneskelig tænkning og gjorde det baseret på den såkaldte λ-calculus, som blev introduceret af logikeren Alonzo Church (1903–1995) i 1930'erne for at besvare nogle af de samme spørgsmål, som vi har diskuteret i relation til Alan Turing (1912–1954) i kapitel 3.

4. Det fjerde af disse klassiske programmeringssprog tog sit udgangspunkt et helt andet sted. COBOL blev udviklet af det amerikanske forsvar i samarbejde med industrien som en bestræbelse på at gøre programmering til en aktivitet, der ikke krævede *særligt* trænede specialister, men kunne udføres og især forstås af det etablerede arbejds- og kommandohierarki (Sammet, 1978). COBOL er et akronym, som står for COmmon Business-Oriented Language, og det var designet, så det lignede naturligt engelsk langt mere end de meget tekniske notationer i mange andre programmeringssprog. COBOL fandt stor udbredelse i bl.a. bank- og finansverdenen, og der er stadig mange vigtige systemer, der er skrevet i COBOL og kører på bagudkompatible installationer, og en ny specifikation af sproget udkom så sent som 2002, selvom sproget ikke længere indgår i de store IT-uddannelser.

Ved starten af 1960'erne havde man altså mindst fire store og blivende aktører på 'markedet' for programmeringssprog: FORTRAN, ALGOL, LISP og COBOL. Hvert af disse sprog havde vægtet forskellige kriterier i deres design, og hvert af dem fandt stor udbredelse i sin egen niche (Nofre, Priestley og Alberts, 2014). Fremvæksten af forskellige sprog var så bemærkelsesværdig, at forsiden af tidsskriftet *Communications of the ACM* i 1961 viste programmeringssprogenes babelstårn. Med henvisning til den bibelske fortælling tegnede tårnet et billede af forfængelighed (hurtig udvikling), som gik ud over forståelsen sprogene imellem. Og — måske — lå der også under illustrationen en normativ påstand om, at et (eller i hvert fald en lille håndfuld) programmeringssprog burde være tilstrækkeligt.

Babelstårnet

Men sådan skulle det ikke gå. I 1972 kunne datalogen Jean E. Sammet (1928–2017) se tilbage på et årti, hvor mængden og variationen af programmeringssprog var eksploderet. Mange nye sprog var kommet til, men af det oprindelige babelstårn fra 1961 stod kun 10 tilbage som en stige til at understøtte det væltende tårn (Sammet, 1972, s. 42), heriblandt de fire, vi har beskrevet ovenfor (se også Ensmenger, 2010). Men også nye vigtige ideer var kommet til at yde indflydelse på design af programmeringssprog (se fortsat figur 2):

5. Abstraktionen fra hardwarens opbygning i retning af programmørens behov tog et stort skridt med programmeringssproget SIMULA, som blev udviklet af Ole-Johan

Dahl (1931–2002) og Kristen Nygaard (1926–2002) ved *Norsk Regnesentral* (Norwegian Computing Centre), som var under (delvis) kontrakt fra det norske forsvar (Holmevik, 1994). SIMULA var — som navnet antyder — beregnet til at simulere processer, hvor forskellige *agenter* interagerer med hinanden. Det kan være ventende passagerer ved et busstop, men det kunne også være elementarpartikler i en nuklear reaktion. Dertil designede Dahl et *objekt-orienteret* programmeringssprog, hvor hver agent kunne tænkes som et stykke selvstændig kode, der indkasplede både data og kommandoer (Krogdahl, 2005). Og med SIMULA-67 blev sproget forbundet til ALGOL og kom til at udgøre en både teoretisk og praktisk vigtig brik blandt programmeringssprog.

6. Hvor mange af de udviklede nye programmeringssprog havde bestemte anvendelser i sigte, var der også nogle sprogudviklere, der fokuserede på undervisningsbehov. Et af de sprog, der kom ud af den tradition var BASIC, som blev udviklet på det meget progressive Dartmouth College af John G. Kemeny (1926–1992) og Thomas E. Kurtz i 1964. Formålet var at gøre programmering tilgængeligt for de universitetsstuderende, og derfor havde BASIC, som står for 'Beginners' All-purpose Symbolic Instruction Code' et let tilgængeligt instruktionssæt, som nøje svarede til de simpleste algoritmiske kontrolstrukturer med loops og branches. Rækkefølgen var bygget op omkring linjenumre, så **go to** var allestedsværende, og omfaktorisering var en væsentlig forhindring, da der typisk ikke var indbygget mulighed for at justere linjenumre og skaffe mere 'plads' et sted i koden. Til gengæld var sproget let at implementere, og det vandt bl.a. bred udbredelse som det indbyggede programmeringssprog i den uhyre populære *Commodore 64*.

Denne blok af programmeringssprog var altså blevet tilføjet 'babelstårnet', da NATO-konferencen i 1968 identificerede nogle af de presserende behov inden for softwareudvikling i de kommende år. Blandt andet pegede man på, at der skulle uddannes mange flere programmører, og at deres uddannelse skulle sikre, at de bidrog til at skrive pålideligt software. Samtidig pegede mange af de nævnte dataloger på konferencen også på, at *programmeringssprog* kunne være en understøttelse af udviklingen af god software, og særligt, at *struktureret programmering* kunne være vejen frem imod klarere og mere modulariseret software, der var nemmere at teste og ikke blev så kompliceret af et flow, der mest af alt mindede om en portion kogt spaghetti.

7. Et af de andre sprog, som blev udviklet med både *struktureret programmering* og undervisningsbehov for øje var PASCAL, som blev designet af Wirth, som efterfølgende også udviklede andre programmeringssprog fx MODULA-2, og udkom i 1970. Wirth havde deltaget i gruppearbejdet med at udvikle en ny version af ALGOL, men han var blevet frustreret over de mange plenumsdiskussioner og teoretiske uenigheder. I stedet udviklede han PASCAL, som vandt stor udbredelse blandt „computer novices“, der i modsætning til „the high-priests of computing centres“ ikke var bundet af opbygget erfaring og præference for forudgående programmeringssprog (Wirth, 1993, s. 11). PASCAL blev hurtigt udbredt til mange platforme, bl.a. fordi man oversatte kildekoden til en slags halvfabrikata, *p-code*, som enten kunne oversættes videre eller fortolkes.

8. Programmeringssproget PROLOG er det mest udbredte eksempel på et *logisk programmeringssprog*. Selve navnet er igen et akronym, denne gang for 'PROgrammation

en LOGique', og som dette antyder, er ophavet til PROLOG fransk-canadisk (Colmerauer og Roussel, 1993, s. 2). PROLOG var resultatet af arbejde med at bygge en såkaldt *interaktiv prover* til naturlige sprog. Formålet var at få computeren til at 'forstå' og 'ræsonere logisk' omkring fx *syllogismer* (se figur 1 og kapitel 6). Og i 1972 blev PROLOG implementeret i varianten ALGOL-W, som Wirth havde udviklet. Hovedmændene bag designet af PROLOG var Alain Colmerauer (1941–2017) og Robert Kowalski, og styrken af PROLOG skyldes både det interaktive interface og den effektive *heuristiske pruning*, som siden også er blevet udviklet kraftigt.

Input:
```
Every psychiatrist is a person.
Every person he analyses is sick.
Jacques is a psychiatrist in Marseilles.
Is Jacques a person?
Where is Jacques?
Is Jacques sick?
```

Output:
```
Yes.
In Marseilles.
I don't know.
```

Figur 1: Eksempel på en original session i PROLOG (Colmerauer og Roussel, 1993, s. 7).

Selvom der gik en udvikling i retning af at gøre sprogene mere tilgængelige for brugerne, så var der også et modsatrettet behov for programmeringssprog, der kunne levere effektiv og sikker kode til især såkaldt *system software*. Dette omfattede de mest centrale del af et computersystem som fx et *operativsystem*, der står for håndteringen af hukommelse, filer og andre eksterne enheder. Og selvom fx PASCAL også blev brugt til *system software*, så blev denne rolle i høj grad overtaget af andre sprog og især af C.

9. Dennis Ritchie (1941–2011) designede programmeringssproget C i begyndelsen af 1970'erne, mens han arbejdede for *Bell Laboratories*, som er en amerikansk forskningsinstitution, oprindeligt grundlagt af opfinderen Alexander Graham Bell (1847–1922). Ritchie var sammen med Ken Thompson centralt involveret i at designe og implementere UNIX, og C blev det fortrukne programmeringssprog til implementationen, hvilket gjorde UNIX portabelt til mange platforme i takt med, at C-*compileren* blev porteret. C var en videreudvikling af et tidligere programmeringssprog kaldet *B*, og et af de fascinerende elementer i tilblivelsen af disse sprog var, at Thompson var i stand til at implementere en *B-compiler i* sproget *B*. Dermed blev det muligt at 'bootstrappe' compilere ved at implementere en (lille) kerne af sproget i sig selv (Ritchie, 1993, s. 4). Og kombineret med en *preprocessor* og et stadigt voksende antal standardbiblioteker med udvidelser blev C hurtigt til et potent og udbredt programmeringssprog. Det førte også til, at C blev *standardiseret* i 1982, og siden er der kommet forskellige udviklinger i sproget.

10. En af de væsentligste udvidelser af C er det *objekt-orienterede programmeringssprog* C++, som den danske datalog Bjarne Stroustrup designede fra slutningen af 1970'erne. Stroustrup beskriver selv sine designkriterier således:

> C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming. It was intended to deliver that to real projects within half a year of the idea. It succeeded. (Stroustrup, 1993, s. 1)

Stroustrup havde kendskab til SIMULA fra sin uddannelse i Aarhus, og han tilføjede i første omgang klasser til C ved at programmere en *preprocessor*, inden at det egentlige C++ så dagens lys i 1985 og hurtigt blev et meget anvendt værktøj til *objekt-orienteret programmering*.

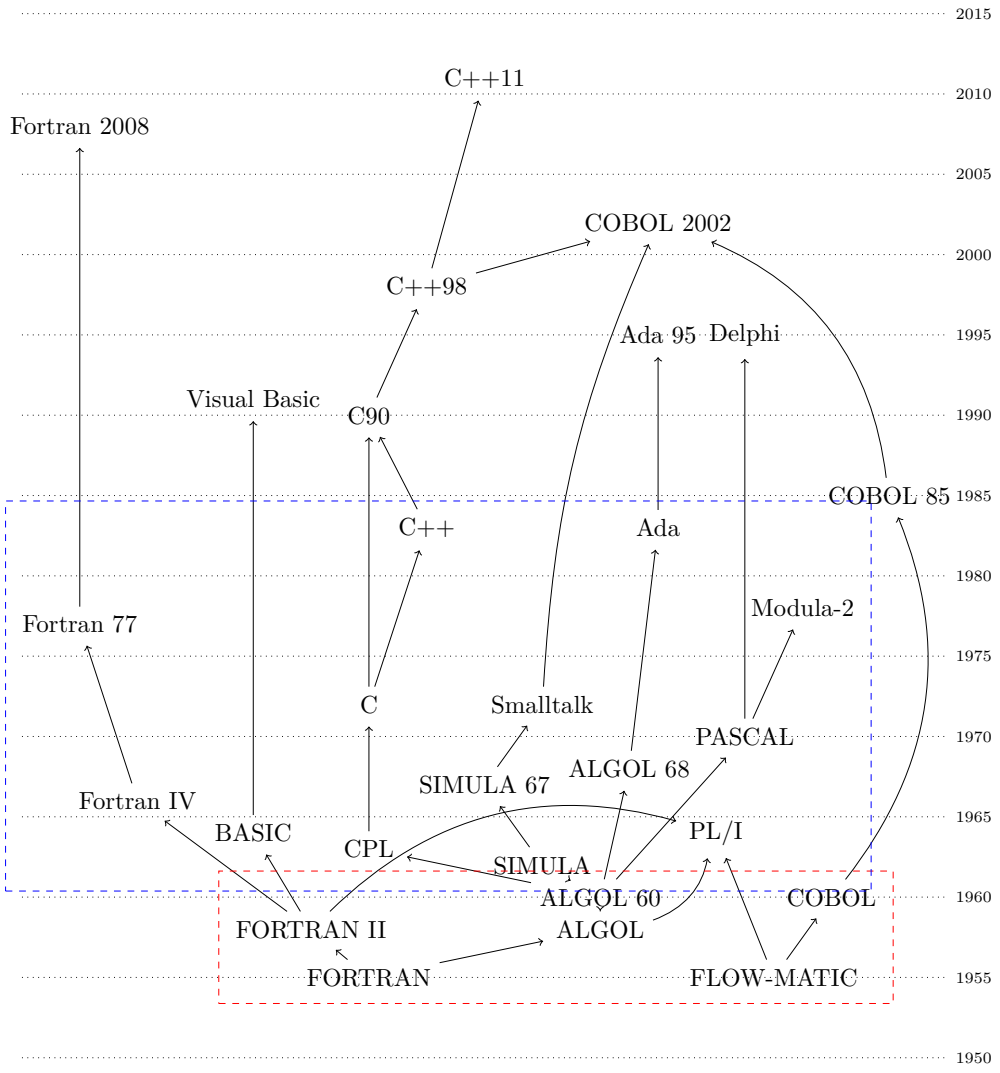
Navn	**Årstal**	**Design**	**Paradigme**
ALGOL	1958	Backus, Naur, Perlis et al	I
BASIC	1964	Kemeny og Kurtz	I
C	1972	Ritchie v/ Bell Labs	P/S
C++	1985	Stroustrup	M/P/S/F/OO
COBOL	1959	US DoD	I/P/(OO)
LISP	1958	McCarthy	M/F/P
FORTRAN	1957	Backus v/ IBM	I
PASCAL	1970	Wirth	I/S
PROLOG	1972	Colmerauer og Kowalski	L
SIMULA	1962	Dahl	M/P/S/OO

Figur 2: Oversigt over nogle af de vigtigste programmeringssprog. Årstal henviser til første design, mange af sprogene er udviklet løbende og er stadig i anvendelse. Forkortelserne er M: multi-paradigmatisk, I: imperativt, P: procedurelt (P $\subset$ I), S: struktureret, F: funktionelt, L: logisk, OO: objekt-orienteret.

Hvad gør et godt sprog?

Ud fra denne korte gennemgang af 10 vigtige programmeringssprog (se også figur 2) kan man gøre sig en række observationer.

For det første kan man se, at den allerførste generation af programmeringssprog som især FORTRAN og ALGOL kom til at udgøre, hvad man kan kalde en *sprogstamme*: Rigtig mange af de efterfølgende programmeringssprog er blevet til som videreudviklinger af disse tidlige programmeringssprog. Man taler endda om FORTRAN-ALGOL-C-familien af programmeringssprog (se figur 3), hvortil mange af de sprog, vi bruger i dag, stadig hører.



Figur 3: Relationerne mellem programmeringssprog i FORTRAN-ALGOL-C-familien. Der er mange flere sprog og sprogvarianter, men de fleste af de vigtigste er med her. Den røde firkant indrammer de store sprog i 1. generation (1950'erne), og den blå firkant indrammer store sprog i 2. generation.

Hvis man følger sprogenes udvikling over tid kan man også se, hvordan de gensidigt påvirker hinanden, så meget, at flere af de nyere sprog er opstået som fusioner mellem tidligere sproggrene. Det er mest tydeligt tilfældet med C++, der eksplicit blev skabt for at kombinere egenskaber fra SIMULA (programorganisering) og C (effektivitet).

Endvidere kan man se, at der har været forskellige forstillinger om sprogenes formål og foretaget forskellige vægtninger imellem hensyn, når nye sprog er blevet designet. Behov i uddannelse var en vigtig motivation for BASIC og PASCAL (Goosen, 2008). Og relationen mellem naturligt sprog og programmeringssprog var en tilbagevendende overvejelse og forskningsinteresse bag såvel COBOL som PROLOG (Good og Howland, 2017).

Et af de vigtige skel i forhold til programmeringssprogs formål handler om vægtningen mellem kontrol og understøttelse for programmøren. Og deri ligger også en god del af diskussionen om programmørens status og relation til brugeren (se også kapitel 7). Man

kan indføre et skel imellem *lavniveausprog*, som er designet til at give programmøren en høj grad af kontrol over, hvordan programmet oversættes til maskinkode i et trade-off mod større besvær med programmeringen. Denne type sprog vil kunne give meget effektiv softwareudførelse, men med større investeringsbehov i udviklingen. Nogle typiske eksempler på *lavniveausprog* er C og FORTRAN. Modsætningen hertil vil være *højniveausprog*, som skjuler mange implementationsdetaljer i et trade-off mod ydelse af koden. Eksempler på denne type sprog er BASIC, LISP og PROLOG. Disse sprog er typisk *fortolkede sprog* og tilbyder fx hukommelseshåndtering i form af *garbage collection*. Jo højere niveauet af programmeringssproget, des sværere bliver det at forudsige ydelsen af den producerede software, hvilket vil være et stort problem for realtids- eller ydelseskritiske systemer.

Afvejningen af hensyn hænger også nært sammen med en afvejning af anvendelsesområder. I en direkte sammenligning af PASCAL og C identificerede Feuer og Gehani (1982) fire typiske anvendelser: 1. „business data processing“, 2. „scientific programming“, 3. „programming of operating systems“, og 4. „programming for system utilities“ (Feuer og Gehani, 1982). På de fleste parametre fandt denne sammenligning, at sprogene var lige (u)egnede til 1, hvor COBOL dominerede, 2, hvor FORTRAN stadig dominerede, og 4. Men på programmering af *operativsystemer*, var C klart det mest effektive, hvilket jo også netop var Ritchies motivation bag sproget.

Hvis man ønsker at karakterisere et succesfuldt sprog kan nogle af de parametre, som eksemplerne ovenfor har antydet, være

- Hvor let det er at *lære* sproget. Det har været et designkriterium for fx BASIC og PASCAL.
- Hvor let det er at *udtrykke* ting i sproget, når man først har lært det — vi taler sommetider om, at sproget er *kraftfuldt*. Det har været et designkriterium for fx C og LISP.
- Hvor let det er at *implementere* sproget, hvilket også kan sige noget om, hvor hurtigt det kan nå udbredelse på forskellige platforme. Det var en del af designsuccessen for BASIC.
- Hvor let det er at *genererere god maskinkode* (målt på fx hurtighed eller størrelse). Det var særligt en motivation for fx FORTRAN.
- Hvor let det er at finde *opbakning* fra vigtige spillere i markedet til sit sprog. En stor del af successen af COBOL kan henføres til dette kriterium, og siden hen har andre sprog som VISUAL BASIC eller JAVA fundet opbakning fra nogle af de helt store software- og hardwareproducenter.
- Hvor let det er at nå nem og billig *udbredelse* til et stort publikum. Her er både PASCAL og JAVA gode eksempler på, at det kan lykkes, hvis man stiller programmeringssproget frit til rådighed.

Nogle af de egenskaber, som sprog værdsætter meget forskelligt er fx *rekursion*, *stakhåndtering*, *statiske typer* og *garbage collection*: Nogle sprog kan ikke fungere uden en af disse, mens andre sprog slet ikke indeholder dem. Og mange diskussioner om programmeringssprogs relative meritter handler om præferencer blandt disse egenskaber, sprogenes syntaks og deres primære anvendelsesområde. To af de væsentligste balancegange, som

sprogdesignere skulle gå mellem hensyn til programmøren og effektivitet i oversættelse og afvikling, var *syntaks* og *typehåndtering*.

For *syntaks* var balancen typisk mellem formel elegance og effektiv evaluering, og mange programmeringssprog opfandt ideosynkratiske syntaktiske konstruktioner. Men alligevel blev noget nedarvet, især igennem *sprogstammen*, så man kan typisk lettere læse og forstå nært beslægtede programmeringssprog — ligesom tilfældet er for naturlige sprog. At syntaks også afspejler en abstraktion fra hardware i retning af programmøren kan man fx se ved at betragte forskellige sprogs syntaks for *increment*: operationen, der lægger 1 til en variabel. FORTRAN, som netop var designet omkring elegant omgang med formler, vil forvente, at man skriver følgende:

```
i = i+1
```

Det er jo super klart, hvad der skal foregå, men det er ikke sikkert, at det giver maksimalt effektiv kode. I C, derimod, opfandt THOMPSON følgende syntaks:

```
i++;
```

Indholdet er det samme, men i C-koden ved *compileren* på forhånd, at der er tale om et *increment*, og den kan oversætte det til den tilsvarende maskininstruktion, der altid findes i instruktionssættet. Det kunne FORTRAN-*compileren* også *udlede*, men her var det givet, og C-syntaksen fjerner jo også mulighed for visse skrivefejl, omend den forvirrer mange, første gang, de ser den.

Dette er et eksempel på, hvordan *syntaks* kan bruges til at understøtte robust og hensigtsmæssig kode. Et eksempel kunne også være følgende lille programstump, der viser en lille, lidt ikke-intuitiv justering af koden, der gør den lidt mere robust.

```
unsigned long factorial(unsigned char n) {
  if (1==n) {
    return 1;
  } else {
    return n*factorial(n-1);
  }
}
```

For hvor lighedstegn jo i C bruges til tildeling, er syntaksen for lighedstest et dobbelt lighedstegn. Dette er måske en uhensigtsmæssig konvention for den uindviede, men kodestumpen gør, at, hvis man ellers skriver sine tests konsekvent på den måde, vil en skrivefejl af `=` for `==` blive fanget af *compileren*.

Den anden væsentlige design-betragtning, som vi vil behandle kort her er *typehåndtering*. Typer er jo ikke noget, der findes på hardware-niveau, så de hører til i den abstraktion, som programmeringssproget stiller til rådighed. Og samtidig viste (statiske) typer sig at være både en velsignelse og en forbandelse for programmøren: Typetjek er gode til at fange mange fejl i programmeringen, men statiske typer er træls at arbejde med, når man fx skriver generiske rutiner til sortering. Og her valgte nogle sprog, som C, så den løsning at stille både typetjek og muligheden for at omgå det til rådighed for programmøren. Deraf kommer meget af C's højt værdsatte effektivitet og fleksibilitet, men også en meget stor del af bugs i C skyldes fejlagtig *recasting* og dårlig håndtering af stakke.

4.3 Programmeringsparadigmer

En måde at organisere de mange programmeringssprog, som er blevet udviklet, på, er at gruppere dem efter deres basale abstraktioner. Det fører til en inddeling i det, man typisk kalder *programmeringsparadigmer*, som vi her kort skal introducere og diskutere.

Man skelner overordnet mellem to typer af programmeringssprog, nemlig *imperative* og *deklarative* sprog (se figur 4). De *imperative* sprog er karakteriseret ved, at man som programmør giver kommandoer, mens programmører, der bruger *deklarative* sprog, i stedet deklarerer antagelsser og mål for afviklingen af programmet. Denne skelnen fortjener at blive udfoldet og forklaret, og det vil vi gøre ved yderligere at underinddele hver kategori.

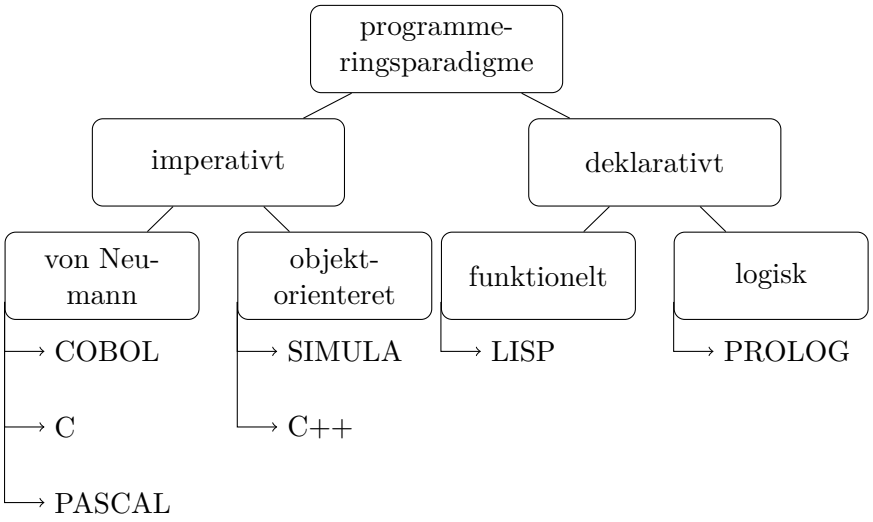
Blandt de *imperative* sprog skelner man typisk imellem *procedurelle* sprog — også kaldet *von Neumann-sprog* — som er den klasse af sprog, der er tættest baseret på hardware-paradigmet (se også kapitel 2). De arbejder altså med en program-tilstand, som ændres over tid i takt med, at kommandoer styrer program-flow og manipulerer hukommelsen. Disse kommandoer udgør kontrolstrukturer og omfatter bl.a. assignment, I/O-operationer og procedurekald. Dermed ligner denne type sprog ganske meget karakteriseringen af *algoritmer* (se kap. 3), og de har *procedurer* som deres naturlige abstraktioner. Mange af de klassiske programmeringssprog, som vi har gennemgået ovenfor, falder i denne kategori, fx COBOL, PASCAL og C. Nogle af de andre sprog er også *imperative* uden at have en så god understøttelse af *modularisering* igennem procedurekald, fx ALGOL, FORTRAN og BASIC.

Den anden store klasse af *imperative* sprog udgøres af de *objekt-orienterede* sprog. Disse sprog er baseret på de måder, relationer består mellem begreber, og på abstraktioner fra den måde, menneskelig kommunikation og interaktion med omverdenen foregår på. De er karakteriseret ved en bestemt form for *modularisering*, hvor både data og operationer er *indkapslet* i *objekter*, hvorved intern information i form af attributter og operationer i form af håndtag er skjult for resten af programmet. I stedet foregår en afvikling af programmet ved, at objekter interagerer igennem beskeder. I *objekt-orienterede sprog* er objekterne typisk beskrevet som instanser af mere generelle klasser (begreber) med mulighed for nedarvning og specialisering. Blandt de klassiske programmeringssprog er især SIMULA og C++ eksempler på *objekt-orienterede programmeringssprog*.

Også iblandt de *deklarative* sprog skelner man typisk mellem to store underklasser. Den ene og mest righoldige underklasse udgøres af *funktionelle* programmeringssprog, som er baseret på funktioner som den naturlige abstraktion. Det betyder, at funktioner er de primære objekter, ligesom tal og lister typisk er det. I modsætning til de *imperative* sprog er *funktionelle* sprog i princippet atemporelle, og afvikling foregår igennem anvendelse (*applicering*) af funktioner. Det oprindelige 1.-generationssprog, hvor det *funktionelle programmeringsparadigme* blev etableret, er LISP, men siden er der kommet mange forskellige varianter til, ligesom flere af de *imperative* sprog også muliggør *funktionel* programmering.

Den sidste vigtige underklasse udgøres af *logiske* programmeringssprog, der er en meget ren form for *deklarative* sprog, baseret på erfaringer fra *ATP*automatiserede bevisgeneratorer (se kap. 3). Disse sprog har, ligesom de *funktionelle sprog*, deres oprindelse i AI, hvor de blev brugt både til modellere logisk ræsonnement og processering af naturlige sprog. De basale entiteter er her aksiomer, inferensregler og forespørgsler fra brugeren, og en afvikling svarer til en systematisk afsøgning fra en mængde facts under brug af aksiomerne og inferensreglerne og en søgeheuristik med effektiv pruning af søgetræet (se også figur 1). Eksemplet på et *logisk programmeringssprog* iblandt de 10 klassiske sprog er PROLOG, men der er inden for de seneste tiår kommet mange andre og meget stærke *automatiske*

bevisgeneratorer på markedet, som også falder i denne kategori.



Figur 4: Skematisk oversigt over de centrale programmeringsparadigmer.

Foruden de 'rene' eksemplarer i hvert programmeringsparadigme, findes der også forskellige hybrider, der kombinerer på tværs af kategorierne. For eksempel er C++ et *multiparadigmatisk sprog*, idet det omfatter både det *imperative* sprog C og *objekt-orientering*. Det samme gør sig gældende for en række af de såkaldte *script-sprog* som fx PERL og PYTHON.

Vi omtaler disse forskellige programmeringsparadigmer netop som *paradigmer*, fordi de ligesom *kuhnske paradigmer* udgør en hel tilgang — en verdensanskuelse (se også kapitel 2). Det er ikke kun et spørgsmål om forskellige syntakser og mindre styrker og svagheder ved det enkelte programmeringssprog, der skelner programmeringsparadigmerne. Det er derimod spørgsmål om hele problem-opfattelsen og problem-løsningen, der ser forskellig ud fra forskellige paradigmers verdenssyn. Hvis man fx *tænker objekt-orienteret*, så *består* den verden, vi skal modellere med computersystemer, af objekter, som er instanser af begreber (klasser). Programmeringsparadigmet er med andre ord det filter, som vi anskuer og modellerer verden igennem. På den måde svarer programmeringsparadigmer jo udmærket til Thomas Kuhns (1922–1996) beskrivelse, og man kan også argumentere for, at dataloger netop igennem deres uddannelse socialiseres ind i en disciplinær matrix for hvert programmeringsparadigme: Der er nogle klassiske og typisk problemer, som de forskellige sprogklasser kan løse, ligesom man igennem lærebøger lærer at anvende typiske teknikker til problemløsningen. Og mange af lærebøgerne og sprogbeskrivelserne har eksemplarisk status og er blevet til små ikoner, som fx Brian Kernighan og Ritchies beskrivelse af C (Kernighan og Ritchie, 1978). Og endelig vil mange programmører måske også opleve noget i retning af en *omvendelse*, når de første gang møder et programmeringsparadigme, der rækker ud over det *procedurelle*. Dermed er der mange lighedspunkter, både i det altomfattende verdenssyn, den sociale disciplinering og den psykologiske oplevelse, som berettiger og måske fordrer, at vi sammenligner programmeringsparadigmer med *kuhnsk*kuhnske paradigmer.

Men der er også nogle vigtige forskelle, som har med programmeringens særlige status som vidensfelt at gøre. For alle disse programmeringsparadigmer er *sameksisterende*. Og selvom det kan være svært at vænne sig til et andet programmeringsparadigme, er der fuld

oversættelighed imellem dem, både teoretisk idet alle sprogene har præcist samme udtryks-
kraft (de er alle *Turing-komplette sprog*) og fordi de alle afvikles på sammenlignelige og
emulerbare arkitekturer. Så hvor Kuhn forestillede sig, at paradigmer (i *realvidenskaberne*)
er *inkommensurable*, så er programmeringssprog — i kraft af, at de er formelle systemer,
som mennesker har bestemt — anderledes stedt. For inden for *formalvidenskaberne* er der
nærmere tale om, at paradigmet definerer *vores* måde at anskue verden på, når vi foretager
modellering og er klar over, at vores perspektiv er ufuldstændigt.

Alligevel kan der være god grund til at reflektere over programmeringsparadigmer som verdensanskuelser (Wernick og Hall, 2004). For det første sætter det tydeligt fokus på, at forskellige paradigmer har forskellige styrker — og at disse styrker skal ses i forhold til programmeringen og ikke i forhold til virkeligheden. For det andet viser parallellen os, at der altid er visse elementer, vi ikke kan medtage i vores *modellering* (se også kapitel 5), hvorfor vi gør klogt i at reflektere over vores valg af paradigme og dets relative blinde vinkler.

4.4 Fra algoritme til afvikling

Det betaler sig — både fra et datalogisk og et videnskabsteoretisk standpunkt — at have et billede af den proces, hvormed algoritmer kan omsættes til afvikling på en computer (se også Bryant og O'Hallaron, 2011).

Lad os forestille os, at vi har følgende opskrift:

Algoritme 4.1
For at beregne $\frac{c(c+1)}{2}$ skal du lægge tallene fra 1 til c sammen.

Denne opskrift ligner det, som Robin K. Hill karakteriserede som en algoritme (se kapitel 3), ved at den opfylder at være 1. *endelig* (den er skrevet op på mindre end to linjer her), 2. *abstrakt* (den gælder ikke kun for fx $c = 4$, men for generelle c), 3. *effektiv* (den kan udføres mekanisk, i hvert fald, hvis addition ikke kræver mere end 'tankeløs handlen'), 4. *sammensat kontrolstruktur* (den giver instruktioner om sekvensen af operationer, der hver især er sammensatte) og 5. givet *imperativt* (hver af operationerne er givet som ordrer til udførelse). Desuden har opskriften et 6. *formål* (nemlig at udregne $\frac{c(c+1)}{2}$), men 7. den har ikke eksplicit angivne *betingelser*. Dermed opfylder opskriften alle kriterier i Hills første definition af en algoritme, men den mangler at opfylde et enkelt i hendes anden, mere raffinerede definition.

Opskriften er endvidere en *korrekt* algoritme, hvilket man kan bevise ved et simpelt matematisk induktionsbevis over c, der beviser ligheden

$$VS(c) := \quad \frac{c(c+1)}{2} = \sum_{x=1}^{c} x \quad =: HS(c). \tag{1}$$

Som induktionsstart indser vi, at for $c = 1$ er $\frac{1(1+1)}{2} = 1$. Og som induktionsskridt kan vi

opskrive

$$VS(c+1) = \frac{(c+1)((c+1)+1)}{2} = \frac{c(c+1)+2(c+1)}{2} = \frac{c(c+1)}{2} + (c+1)$$
$$= \sum_{x=1}^{c} x + (c+1) = \sum_{x=1}^{c+1} x = HS(c+1),$$

hvor linjeskiftet svarer til brug af induktionshypotesen.

Denne opskrift kan implementeres på mange forskellige måder og i mange forskellige formalismer og programmeringssprog. Lad os betragte følgende lille stump C-kode til beregning af højresiden i (1):

Kildekode 4.1: En implementation af algoritme 4.1

```
x=c; y=0;
while (x>0) {
  y+=x; x--;
}
```

Denne kode kunne umiddelbart se ud til at være en korrekt implementation af algoritme 4.1: For hvert gennemløb af while-loopet forøges `y` med værdien af `x`, hvorefter `x` tælles ned fra `c` til 0, så når vi når slutningen, ser `y` ud til at være summen af `x` for `x` mellem 1 og `c`, og altså `y==c(c+1)/2`.

Men hvad nu, hvis `c=-1` ved starten af programstumpen? Så bliver loop-betingelsen aldrig opfyldt, og vi får `y==0` ved slutningen af koden. Dette kunne sådan set godt anses som korrekt opførsel, idet formlen stadig er opfyldt. Men hvis man havde sat `c=-2;` ved starten af koden, havde man stadig fået `y==0`, og det er klart, at koden ikke altid giver korrekte resultater for negative værdier af `c`. Men faktisk er den indvending ikke så meget rettet mod kildekode 4.1 som imod algoritme 4.1, hvor vi netop ikke havde opstillet nogen betingelser på c. Og antagelsen ligger faktisk implicit i formel (1). Så for at algoritme 4.1 virkelig skal opfylde den raffinerede definition hos Hill, skal der netop gøres den antagelse, at c er et positivt (hel)tal.

Hvis vi vender tilbage til figur **??** (se figur 6, nedenfor), beskrev den, hvordan et program som kildekode 4.1 (en implementation i kildekode) er en specificering af den abstrakte algoritme til et konkret programmeringssprog (1). Og en af egenskaberne ved C er, at det er et typet sprog med forskellige heltalstyper. Specielt er der ikke i C indbygget understøttelse af vilkårligt store tal, så allerede med valget af programmeringssprog indtræder en begrænsning: Der vil være værdier af $\frac{c(c+1)}{2}$, som ikke kan repræsenteres i de indbyggede tal-typer.

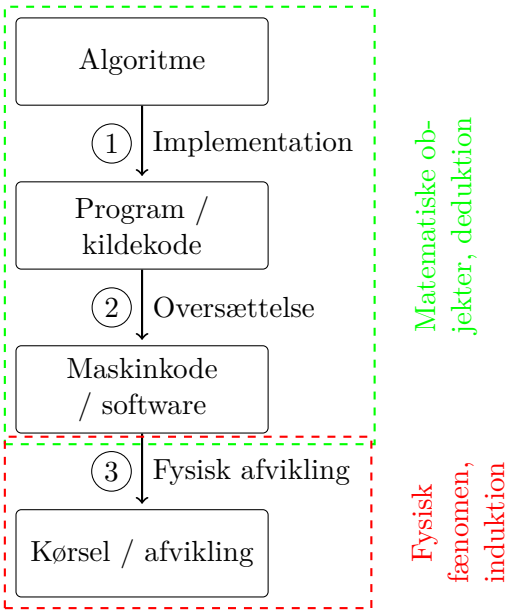
Den næste specificering (2) i figur 6 handler om, at oversættelsen fra kildekode til maskinkode er betinget af den arkitektur, der oversættes til. Så hvordan ville programmets betydning se ud, hvis der inden starten af stumpen var foretaget en erklæring som `int x,c=16; char y;` og jeg ønskede at compile programmet til min x86-arkitektur? Så vil der ske et overrun i `y`, fordi en `char` ifølge specifikationen af C er en byte-variabel, som på denne arkitektur er 8-bit og kun kan indeholde heltal i intervallet $[-128, 127]$. Vi vil derfor få `y==-120`, selvom `c*(c+1)/2==136`. Denne fejl skyldes altså en begrænsning, som *compileren* og den underliggende arkitektur lægger ned over problemet (her i form af et overrun), så maskinkoden er altså ikke en korrekt implementation af programmet og algoritmen uden tilføjelse af yderligere betingelser. Og hvis min arkitektur havde været en anden, havde problemet ligget et andet sted (men stadig eksisteret); figur 5 viser, hvordan størrelsen af

en `char` varierer mellem forskellige arkitekturer anno 1978 — i dag er den i praksis fastsat til 8-bit.

	DEC PDP-11	Honeywell 6000	IBM 370	Interdata 8/32
	ASCII	ASCII	EBCDIC	ASCII
`char`	8 bits	9 bits	8 bits	8 bits
`int`	16	36	32	32
`short`	16	36	16	16
`long`	32	36	32	32
`float`	32	36	32	32
`double`	64	72	64	64

Figur 5: Størrelse af forskellige variabel-typer på forskellige fremherskende 1978-arkitekturer som beskrevet i Kernighan og Ritchie (1978, s. 42).

Endelig viser (3), at maskinkoden skal bringes til fysisk afvikling på en konkret computer. For at denne kørsel af maskinkoden skal give korrekte resultater, forstået som resultater, der er i overensstemmelse med algoritmens specifikation, er det selvfølgeligt afgørende, at de ovenfor omtalte oversættelser (specificeringer), som fører til maskinkoden er korrekte. Men derudover tilstøder der en yderligere komplikation, som hænger sammen med overgangen fra matematiske, immaterielle objekter til afvikling på en fysisk, materiel maskine. For hvor matematikken forholder sig til beviselig korrekthed inden for et *lukket system*, så er afviklingen på en fysisk computer del af et *åbent system*. Det betyder konkret, at vi ikke kan være sikre på, at computeren ikke har produktionsfejl, at tilfældige forbipasserende ikke afbryder beregningen, og at den kosmiske baggrundsstråling ikke påvirker de interne registre og skifter værdien af en bit. Alle disse — og utalligt mange flere — faktorer er ikke modelleret med i vores lukkede system, og vi kan ikke udelukke dem med formelle metoder (deduktive beviser). I stedet må vi forlade os på, at de er sjældne og bruge realvidenskabelige, empiriske argumenter til at overbevise os om det.



Figur 6: Figur **??** med yderligere bemærkninger.

4.5 Fuld fart frem

I august 2009 omkom Mark Saylor (–2009), hans kone, datter og svoger i en trafikulykke i San Diego. Denne ulykke var imidlertid helt særlig af en række omstændigheder: Føreren af bilen, Saylor, var ansat ved California Highway Patrol, der har ansvaret for sikkerheden på statens veje. Bilen, de forulykkede i, var en lånebil af mærket *Toyota Lexus 2009 ES350*. Ulykken skyldtes, at bilen pludseligt accelererede uventet og ustoppeligt, så de bragede ind i autoværnet. Og imens ulykken stod på nåede svogeren at ringe 911, så hele ulykkesforløbet blev optaget på alarmcentralen, og efterfølgende har mange kunnet høre med på Saylor-familiens panik og ulykke. Sagen opnåede hurtigt stor offentlig bevågenhed, dels på grund af de dramatiske kilder, dels fordi denne slags ustoppelig acelleration ikke var helt uhørt, og dels fordi føreren af bilen var en ansvarlig og bilkyndig myndighedsperson.

Faktisk havde det relevante bureau i US, National Highway Traffic Safety Administration (NHTSA), længe oplevet problemer med ustoppelig acceleration (Anderson, 2016). Og i 1986 havde NHTSA foretaget en undersøgelse, der slog fast, at denne slags fejl skyldtes, at føreren havde fejlbetjent bilen og i panik og forvirring havde hamret speederen i bund i stedet for at stå på bremsen. Denne forklaring var blevet brugt til at bortforklare sporadiske uheld og en række mere sammenhængende tilfælde med biler af Audi-mærket. Men i 2009 eksploderede antallet af indberetninger om problemer med Toyotaer, særligt Lexus ES350; og mange flere af tilfældene omfattede personskade og endda død som i Saylors sag. Det var nu en ganske alvorlig ting, som både kunne føre til omfattende tilbagetrækning af modeller fra Toyota og til omfattede søgsmål om erstatning og evt. ansvarsplacering (Finch, 2010).

Men før man kunne komme ret meget længere i sagen, var man nødt til at finde ind til, hvori fejlen bestod: kunne den reproduceres og kunne den forklares? Og i det amerikanske system var der tre forskellige aktører, der indgik i denne fejlfinding: Toyota, NHTSA og repræsentanterne for de skadeslidte, der lagde sag an mod Toyota. På grund af den pludselige stigning i antallet af tilfælde og med det udgangspunkt, at panik-hypotesen ikke virkede særligt overbevisende eller betryggende, rettede noget af fokus sig på det forhold, at alle moderne biler nu brugte et Electronic Throttle Control System (ETCS), som altså er en kombination af hardware og software, der ligger 'imellem' førerens kontrol og bilens opførsel. Alle havde de deres eget udgangspunkt: Toyota havde jo fuld kendskab til bilernes mekanik, hardware og software, men havde ikke interesse i at dele denne viden alt for åbent. NHTSA havde ansvaret for at teste bilerne, men havde allerede i 1986 lagt sig fast på en bestemt forklaring, nemlig førerfejl. Og sagsøgerne havde næsten ingen adgang til hardware og software, men havde en klar hypotese om, at fejlen skyldtes ETC-systemet.

I forbindelse med en sag om endnu en ulykke i Oklahoma i 2013 blev datalogen Phil Koopman engageret som ekspertvidne for sagsøgerne i en erstatningssag imod Toyota (Cummings, 2016). Og Koopmans argument for retten er en af de få kilder, vi har til sagens tekniske dimensioner, for efterfølgende er der blevet indgået fortrolighedsklausuler i de kompromisser, der blev indgået.

I forbindelse med sagerne blev National Aeronautics and Space Administration (NASA) bedt om at lave en test af biler af samme mærke, for at forsøge at finde frem til fejlen. I rapporten, som NASA udarbejdede i 2011, kan man blandt andet læse:

> Due to system complexity […] and the many possible electronic hardware and software system interactions, it is not realistic to attempt to 'prove' that the [Electronic Throttle Control System] cannot cause [Unintended Accelerations]. Today's vehicles are sufficiently complex that no reasonable amount of analysis or testing can prove electronics and software have no errors. Therefore absence of proof that the ECTS-I caused a UA does not vindicate the system. (Citeret fra Anderson, 2016, s. 1428)

Og endvidere:

> Proof of the hypothesis that the Electronic Throttle Control System] caused the large throttle opening [Unintende Acceleration]s as described in submitted [Vehicle Owner Questionnaires] could not be found with the hardware and software testing performed. […] Because proof that the ETCS-i caused the reported UAs was not found does not mean it could not occur. However, the testing and analysis described in this report did not find that [ETCS] electronics are a likely cause of large throttle openings as reported in the VOQs. (Citeret fra Koopman, 2014)

Men den amerikanske transportministers udlægning af NASA's ekspertvurdering er helt modsat. Ray LaHood sagde i den tilhørende pressemeddelelse:

> We enlisted the best and brightest engineers to study Toyota's electronics systems, and the verdict is in. There is no electronic-based cause for unintended high-speed acceleration in Toyotas. (*U.S. Department of Transportation Releases Results from NHTSA-NASA-Study ofUnintended Acceleration in Toyota Vehicles* 2011)

Fra Koopmans analyse får man et indtryk af kompleksiteten af den software, der ligger i ETCS: Den består af omkring 256.600 linjer aktiv C-kode (excl. kommentarer) og omkring 39.500 headers. Koden afvikles dels på en primær CPU og en proprietær Monitor Chip med tilhørende proprietær software. Dertil kommer, at softwaren ikke i praksis udføres lineært, idet den er ansvarlig for handlingsbaserede forstyrrelser, fx i forbindelse med registrering af pedalernes tilstand og et fail-safe overvågningssystem, der er indbygget for at monitorere det primære system og gribe ind, hvis noget går galt. Professor Koopman pegede i sit indlæg på en række potentielt farlige problemer med den software, som han altså ikke kunne komme til at studere i alle detaljer. Blandt de primære mistanker var en stribe klassiske syndere i kritiske systemer: Problemer omkring samtidighed (fx race-problemer), overskrivning af globale variable, og stack overflows.

Som både NASA og Koopman sådan set var enige om, så kunne ingen mængde afprøvning fritage ETCS for mistanken om fejl. Men omvendt havde man heller ikke kunnet *påvise* fejl af den kritiske type i nogle af de kontrollerede forsøg, man havde anstillet med andre biler af samme mærke. Men disse eksperimenter havde jo — trods forsøg på at gøre dem så realistiske som muligt — ikke været gjort 'in the wild'. Så hvad skal man så stille op?

Der er nok (mindst) to perspektiver at se på dette spørgsmål ud fra. Det ene er et juridisk perspektiv: I retssagen skulle det *bevises*, at producenten var skyld i dødsfaldene, og det var der trods alt ikke nogen 'smoking gun', der beviste. Og på den måde kom adskillige bevisbegreber i spil i retssagen: et matematisk og absolut bevisbegreb, et begreb

om formel verifikation af software og hardware, og det juridiske begreb, der i USA er baseret på 'beyond reasonable doubt' (se også MacKenzie, 2004). Retssagerne endte i stedet med, at der blev indgået hemnmelige forlig med tavshedsklausuler, som gør, at vores viden stadig er begrænset. Men man kan godt antage, at Toyota så selv betragtelige forlig som en mere kosteffektiv løsning på sagen end omfattende og højtprofilerede tilbagetrækninger af modeller fra hele verden (Gokhale, Brooks og Tremblay, 2014).

Men der er også et datalogisk perspektiv at antage, som handler om, hvordan man sikrer sine kritiske systemer mod uforudsete fejl. Og her er der to oplagte muligheder: Vi kan enten forsøge at *bevise* korrektheden af vores kritiske systemer igennenm formel verifikation. Men det er oftest meget omkostningsfuldt, og vi løber alligevel ind i de begrænsninger, som vi allerede har diskuteret i forhold til overgangen fra lukkede til åbne systemer. Eller vi kan forsøge at underkaste vores kritiske systemer omfattende, systematisk og teoridreven afprøvning. Og her når vi så, trods transportministerens naive logiske fejlslutning, til erkendelse af, at vi ikke kan være *sikre* på, at vi har fanget alle potentielle fejl og problemer. Derfor må vi gå en mellemvej, som vi skal se nærmere på i kapitel **??** og tænke korrekthed og afprøvning ind i hele udviklingsforløbet, således at vi kan tage forbehold for den slags fejl, vi allerede kender (race-conditions, overflow, skrivbare globale variable, covert channels, …) og have en god udviklings-, programmerings- og dokumentationskultur, således at vi kan stille os op som eksperter og forsvare, at koden er så tæt på fejlfri, som det er *menneskeligt* muligt under de givne betingelser. Det er nemlig det bedste, vi kan håbe på.

Litteratur

Anderson, Antony F. (2016). „Case Study. NHTSA's Denial of Dr Raghavan's Petition to Investigate Sudden Acceleration in Toyota Vehicles Fitted With Electronic Throttles". *IEEE Access*, bd. 4, s. 1417–1433. DOI: 10.1109/access.2016.2545713.

Bond, Gregory W. (aug. 2005). „Software as Art". *Communications of the ACM*, bd. 48, nr. 8, s. 118–124.

Bryant, Randal E. og David Richard O'Hallaron (2011). *Computer Systems. A Programmer's Perspective*. 2. udg. Addison-Wesley.

Colmerauer, Alain og Philippe Roussel (1993). „The birth of Prolog. Drawn from the [Second] ACM SIGPLAN History of Programming Languages Conference, April 20–23, 1993, HOPL-II". I: New York: ACM Press, s. 37–52. DOI: 10.1145/154766.155362.

Cummings, David M. (dec. 2016). „Embedded Software Under the Courtroom Microscope. A Case Study of the Toyota Unintended Acceleration Trial". *IEEE Technology and Society Magazine*, bd. 35, nr. 4, s. 76–84. DOI: 10.1109/mts.2016.2618681.

Dijkstra, Edsger W. (1968). „Go To Statement Considered Harmful". *Communications of the ACM*, bd. 11, nr. 3, s. 147–148. DOI: 10.1145/362929.362947.

Ensmenger, Nathan (2010). *The Computer Boys Take Over. Computers, Programmers, and the Politics of Technical Expertise*. Cambridge & London: MIT Press.

Feuer, Alan R. og Narain H. Gehani (mar. 1982). „Comparison of the Programming Languages C and Pascal". *ACM Computing Surveys (CSUR)*, bd. 14, nr. 1, s. 73–92. DOI: 10.1145/356869.356872.

Finch, Joel (2010). „Toyota Sudden Acceleration. A Case Study of the National Highway Traffic Safety Administration — Recalls for Change". *Loyola Consumer Law Review*, bd. 22, nr. 4, s. 472–496.

Gokhale, Jayendra, Raymond M. Brooks og Victor J. Tremblay (nov. 2014). „The effect on stockholder wealth of product recalls and government action. The case of Toyota's accelerator pedal recall". *The Quarterly Review of Economics and Finance*, bd. 54, nr. 4, s. 521–528. DOI: 10.1016/j.qref.2014.06.004.

Good, Judith og Kate Howland (apr. 2017). „Programming language, natural language? Supporting the diverse computational activities of novice programmers". *Journal of Visual Languages & Computing*, bd. 39, s. 78–92. DOI: 10.1016/j.jvlc.2016.10.008.

Goosen, Leila (2008). „A Brief History of Choosing First Programming Languages". I: *History of Computing and Education 3 (HCE3)*. Red. af John Impagliazzo. Springer US, s. 167–170. DOI: 10.1007/978-0-387-09657-5.

Haigh, Thomas (jan. 2002). „Software in the 1960s as Concept, Service and Product". *IEEE Annals of the History of Computing*, bd. 24, nr. 1, s. 5–13. DOI: 10.1109/85.988574.

Holmevik, Jan Rune (1994). „Compiling SIMULA: A historical study of technological genesis". *IEEE Annals of the History of Computing*, bd. 16, nr. 4, s. 25–37. DOI: 10.1109/85.329756.

Kernighan, Brian W. og Dennis M. Ritchie (1978). *The C Programming Language*. 1. udg. Prentice Hall PTR.

Klein, Gerwin m.fl. (2018). „Formally Verified Software in the Real World". *Communications of the ACM*, bd. 61, nr. 10, s. 68–77. DOI: 10.1145/3230627.

Koopman, Phil (sep. 2014). *A Case Study of Toyota Unintended Acceleration and Software Safety*. URL: https://users.ece.cmu.edu/~koopman/toyota/koopman-09-18-2014_toyota_slides.pdf (bes. 13.07.2020).

Krogdahl, Stein (2005). „The birth of Simula". I: *History of Nordic Computing*. Springer, s. 261–275. DOI: 10.1007/0-387-24168-X_24.

MacKenzie, Donald (2001). „A View from the Sonnenbilch: On the Historical Sociology of Software and System Dependability". I: *History of Computing: Software Issues*. Red. af Ulf Hashagen, Reinhard Keil-Slawik og Arthur Norberg. International Conference on the History of Computing, ICHC 2000. April 5–7, 2000, Heinz Nixdorf MuseumsForum, Paderborn, Germany. Berlin etc.: Springer, s. 97–122.

— (2004). „Computers and the Sociology of Mathematical Proof". I: *New Trends in the History and Philosophy of Mathematics. New Trends in the History and Philosophy of Mathematics*. Red. af Tinne Hoff Kjeldsen, Stig Andur Pedersen og Lise Mariane Sonne-Hansen. University of Southern Denmark Studies in Philosophy 19. Odense: University Press of Southern Denmark, s. 67–86.

Nofre, David (2010). „Unraveling Algol. US, Europe, and the Creation of a Programming Language". *IEEE Annals of the History of Computing*, bd. 32, nr. 2, s. 58–68. DOI: 10.1109/MAHC.2010.4.

Nofre, David, Mark Priestley og Gerard Alberts (2014). „When Technology Became Language. The Origins of the Linguistic Conception of Computer Programming, 1950–1960". *Technology and Culture*, bd. 55, nr. 1, s. 40–75. DOI: 10.1353/tech.2014.0031.

Northover, Mandy m.fl. (2008). „Towards a Philosophy of Software Development: 40 Years after the Birth of Software Engineering". *Journal for General Philosophy of Science*, bd. 39, nr. 1, s. 85–113. DOI: 10.1007/s10838-008-9068-7.

Ritchie, Dennis M. (1993). „The development of the C language. Drawn from the [Second] ACM SIGPLAN History of Programming Languages Conference, April 20–23, 1993, HOPL-II". I: New York: ACM Press, s. 201–208. DOI: 10.1145/154766.155580.

Rutishauser, Heinz (1967). *Description of ALGOL 60*. Die Grundlehren der mathematischen Wissenschaften 135. New York: Springer-Verlag. DOI: 10.1007/978-3-642-86934-1.

Sammet, Jean E. (jul. 1972). „Programming languages. History and future“. *Communications of the ACM*, bd. 15, nr. 7, s. 601–610. DOI: 10.1145/361454.361485.

Sammet, Jean E. (aug. 1978). „The early history of COBOL“. *ACM SIGPLAN Notices*, bd. 13, nr. 8, s. 121–161. DOI: 10.1145/960118.808378.

Stroustrup, Bjarne (1993). „A history of C++. Drawn from the [Second] ACM SIGPLAN History of Programming Languages Conference, April 20–23, 1993, HOPL-II“. I: New York: ACM Press, s. 271–297. DOI: 10.1145/154766.155375.

U.S. Department of Transportation Releases Results from NHTSA-NASA-Study ofUnintended Acceleration in Toyota Vehicles (feb. 2011). NHTSA. URL: https://one.nhtsa.gov/About-NHTSA/Press-Releases/2011/ci.U.S.-Department-of-Transportation-Releases-Results-from-NHTSA%E2%80%93NASA-Study-of-Unintended-Acceleration-in-Toyota-Vehicles.print (bes. 13.07.2020).

Vilstrup, Helle (1963). *Lærebog i GIER-Algol*. København: Akademisk Forlag.

Wernick, Paul og Tracy Hall (2004). „Can Thomas Kuhn's paradigms help us understand software engineering?“ *European Journal of Information Systems*, bd. 13, s. 235–243.

Wirth, N. (1993). „Recollections about the development of Pascal. Drawn from the [Second] ACM SIGPLAN History of Programming Languages Conference, April 20–23, 1993, HOPL-II“. I: New York: ACM Press, s. 333–342. DOI: 10.1145/154766.155378.

Wirth, Niklaus (2008). „A Brief History of Software Engineering“. *IEEE Annals of the History of Computing*, bd. 30, nr. 3, s. 32–39. DOI: 10.1109/MAHC.2008.33.

Navneliste

Backus, John Warner [John] (1924–2007), 3, 7
Bell, Alexander Graham (1847–1922), 6

Church, Alonzo (1903–1995), 4
Colmerauer, Alain (1941–2017), 6, 7

Dahl, Ole-Johan (1931–2002), 5, 7
Dijkstra, Edsger Wybe [Edsger] (1930–2002), 2

Hill, Robin K., 13, 14

Kernighan, Brian Wilson [Brian], 12
Knuth, Donald E. [Donald], 1
Koopman, Phil, 16, 17
Kowalski, Robert, 6, 7
Kuhn, Thomas Samuel [Thomas] (1922–1996), 12, 13

LaHood, Ray H. [Ray], 17

McCarthy, John (1927–2011), 4, 7
Minsky, Marvin Lee [Marvin] (1927–2016), 4

Naur, Peter (1928–2016), 3, 7
Nygaard, Kristen (1926–2002), 5

Perlis, Alan Jay [Alan] (1922–1990), 3, 7

Ritchie, Dennis MacAlistair [Dennis] (1941–2011), 6, 7, 9, 12

Sammet, Jean E. (1928–2017), 4
Saylor, Mark (–2009), 16
Stroustrup, Bjarne, 7

Thompson, Kenneth Lane [Ken], 6, 10
Turing, Alan Mathison [Alan] (1912–1954), 4

Wirth, Niklaus Emil [Niklaus], 3, 5–7

Indeks

AI, 11
ALGOL, 2–8, 11
algoritme, 11
applicering, 11
assembler, 1
automatiseret bevisgenerator, 11

B, 6
BASIC, 5, 7–9, 11
Bell Laboratories, 6

C, 6–12, 14, 17
C++, 7, 8, 11, 12
COBOL, 4, 7–9, 11
Commodore 64, 5
compiler, 6, 10, 14
CUDOS-normerne, 3

DASK, 3
deklarative programmeringssprog, 11

Electronic Throttle Control System (ETCS), 16
ETCS, *Se* Electronic Throttle Control System

formalvidenskab, 13
fortolkede sprog, 9
FORTRAN, 3, 4, 7–11
funktionelle sprog, 11
funktionelt programmeringssprog, 4, 11

garbage collection, 9
GIER, 3

heuristisk pruning, 6
højniveausprog, 9

IBM, *Se* International Business Machines
imperative programmeringsprog, 11, 12
indkapsling, 11
inkommensurabel, 13
inkommensurable paradigmer, 13
interaktiv prover, 6
International Business Machines (IBM), 3

JAVA, 9

kuhnsk paradigme, 12

lavniveausprog, 9
LISP, 4, 7, 9, 11
logisk programmeringssprog, 5
logiske, 11

maskinkode, 1
modellering, 13
MODULA-2, 5
modularisering, 2, 11
multiparadigmatisk sprog, 12

NASA, *Se* National Aeronautics and Space Administration
National Aeronautics and Space Administration (NASA), 16
National Highway Traffic Safety Administration (NHTSA), 16
NHTSA, *Se* National Highway Traffic Safety Administration

objekt-orienteret programmering, 5, 7, 12
objekt-orienteret programmeringssprog, 11, 12
objekter, 11
operativsystem, 3, 6, 9

paradigme, 1, 12
  kuhnsk, 12
  programmeringsparadigme, 11
PASCAL, 5–9, 11
PERL, 12
preprocessor, 6, 7
procedure, 11
procedurelle programmeringssprog, 11, 12
programmeringsparadigme, 11
  deklarativt, 11
  imperativt, 11, 12
programmeringssprog, 5
  ALGOL, 2–8, 11
  BASIC, 5, 7–9, 11
  C, 6–12, 14, 17
  C++, 7, 8, 11, 12
  COBOL, 4, 7–9, 11
  deklarativt, 11
  FORTRAN, 3, 4, 7–11
  funktionelt, 4, 11
  højniveau, 9
  imperativt, 11, 12
  JAVA, 9
  lavniveau, 9
  LISP, 4, 7, 9, 11
  logisk, 5
  MODULA-2, 5
  objekt-orienteret, 5, 7, 11, 12
  paradigme, 11
  PASCAL, 5–9, 11
  PERL, 12
  procedurelt, 11, 12
  PROLOG, 5–9, 11
  PYTHON, 12
  SIMULA, 4, 5, 7, 8, 11
  struktureret, 2, 5
  VISUAL BASIC, 9
  von Neumann, 11
PROLOG, 5–9, 11
PYTHON, 12

realvidenskab, 13
recasting, 10
rekursion, 9

script-sprog, 12
SIMULA, 4, 5, 7, 8, 11
software, 1
software engineering, 1
softwarekrise, 2
sprogstamme, 7, 10
stakhåndtering, 9
standardiseret, 6
statiske typer, 9
struktureret programmering, 2, 5
syllogisme, 6
syntaks, 10
system software, 6

Toyota Lexus 2009 ES350, 16
Turing-komplette sprog, 13
typehåndtering, 10

UNIX, 6

videnskab
  formalvidenskab, 13
  realvidenskab, 13
VISUAL BASIC, 9
von Neumann-sprog, 11