Jens Erlandsen
IAML
Njalsgade 96
DK 2300 kbh. S.

GESA, et GEnerelt System til Analyse af naturlige sprog, udformet som et oversætter-fortolker system med virtuel mellemkode.

Parsingsystemer til automatisk analyse af naturlige sprog kan udformes på utallige måder - og det gælder næsten uanset hvilken metode, man vælger for selve parsing-processen.

Det er et større projekt at udforme en parser til et rimelig stort udsnit af naturligt sprog. Som regel vil udformningen af så store systemer ske i flere tempi: Primært sker den i kravspecifikationsfasen, hvor der med udgangspunkt i bl.a. brugerbehov og ind- og uddata opstilles en række krav til systemet; Sekundært i designfasen, hvor systemets struktur, algoritmer og datastrukturer endeligt fastlægges. I designfasen vil det typisk være sådan, at der i forhold til de opstillede krav findes flere løsninger.

I denne artikel vil jeg kort skitsere fire system-modeller, og derefter opridse nogle af de overvejelser i designfasen, der førte til at GESA-systemet blev udformet som et oversætter-fortolker system med virtuel mellemkode. Jeg har altså her anlagt en typisk "system-designer synsvinkel" på udformningsproblematikken, idet jeg har set bort fra alle overvejelser angående systemets sproglige kapacitet.

En mere udførlig beskrivelse af GESA, hvor også disse overvejelser er taget med, er givet i SAML nr. 10.

Skemaet med de fire modeller.

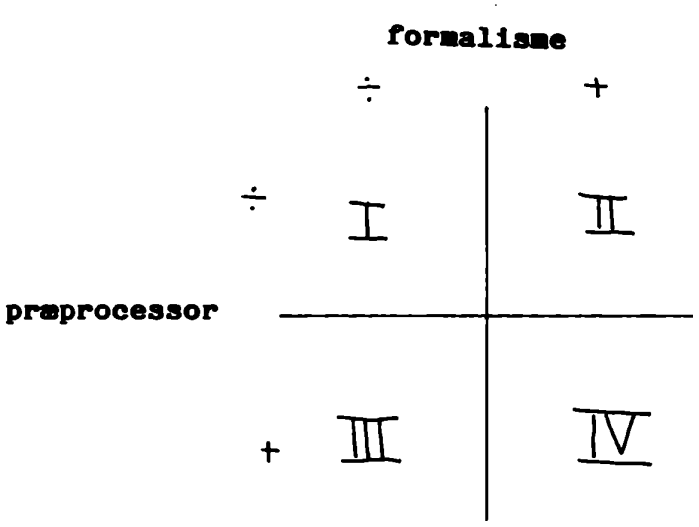
De fire udformningsmåder - eller rettere system-modeller - jeg har valgt at tage med her, kan opstilles i et skema, hvor der er taget hensyn til to forhold:

GESA, et GEnerelt System til Analyse af naturlige sprog, udformet som et oversætter-fortolker system med virtuel mellem-kode
Jens Erlandsen
Proceedings of NODALIDA 1983, pages 74-83

For det første om det samlede parsing-system indeholder en præprocessor i en eller anden form, der behandler sprogbeskrivelsen, inden den anvendes af parseren, eller om det ikke indeholder en sådan processor.

For det andet om grammatikken skal formuleres i en særlig formalisme eller om den formuleres (mere eller mindre) direkte i et allerede kendt programmeringssprog.

Kombineres de to træk, fåes et skema med 4 felter:



MODEL I.

Den første model kan skematisk fremstilles på denne måde:

I denne og i de følgende skitser repræsenterer kasser processorer (dvs. programmer/system-dele) og cirklerne data (fx I: inddata, O: uddata).

I denne model er parser og grammatik bygget sammen i en uadskillelig helhed. Der findes altså ikke nogen særlig præprocessor (programmeringssprogets eventuelle oversætter er ikke medregnet), der behandler en grammatik; og hvis man overhovedet kan tale om en grammatik, er den i hvert fald ikke formuleret i en særlig formalisme (programmeringssproget tæller altså ikke som særlig formalisme i denne sammenhæng).

Er der tale om et eksperimental-system, hvor sprogbeskrivelsen hyppigt ændres og udvides, bliver sådanne systemer let kaotiske og uoverskuelige, hvis ikke man anvender nogle gennemgående principper for data- og process-strukturen.

Et sådant princip kunne fx være recursive-descent (se Aho & Ullman 1977), der kort beskrevet går ud på, at hvert non-terminal i grammatikken (fx på EBNF-form) får sin egen procedure.

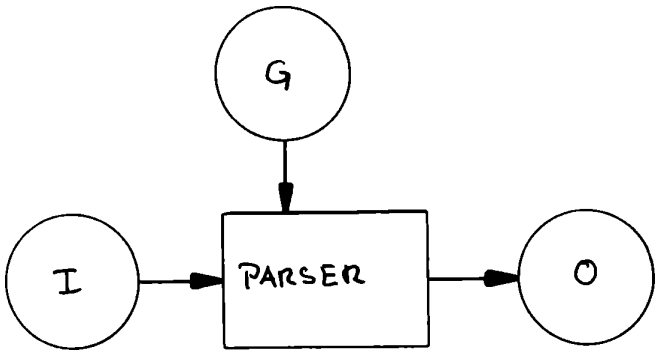
Men selv med anvendelse af sådanne principper er det som regel ikke nogen enkel sag, at ændre sprogbeskrivelsen, fordi det betyder at selve programmet skal ændres. Alle med erfaring i system-arbejde og programmering ved, hvor problematisk det er at ændre større systemer, fx et halvt år efter de er "færdiggjorte".

Har man anvendt et fornuftigt struktureringsprincip, fx det nævnte recursive descent-princip, kan parsere udformet efter denne model blive uhyre effektive.

Så er man i en situation, hvor effektiviteten spiller en stor rolle, fx fordi parseren skal analysere terminal-input fra en bruger i et interaktivt system (fx et undervisningsprogram) og/eller ligger sprogbeskrivelsen helt fast, har modellen måske alligevel en vis berettigelse.

MODEL II.

Denne model, der altså er karakteriseret ved, at grammatikken skrives i en særlig formalisme og direkte i denne form anvendes af parseren under processen, kan skematisk fremstilles på denne måde:



Sådanne systemer kaldes ofte fortolkere, fordi grammatikken fortolkes under processen. Grammatikken betragtes her som data (cirkel G), parseren indlæser før eller under selve parsingprocessen, og den er altså adskilt fra selve parserprogrammet.

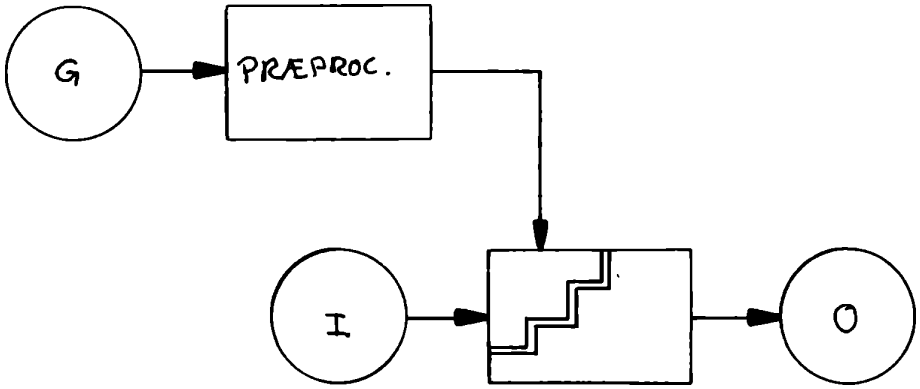
Dette skulle gøre det enklere at ændre og udvide grammatikken, og forandringerne kan måske ligefrem foretages af personer uden kendskab til programmering.

På den anden side er brugervenlige formalismer, og det vil her sige formalismer, hvori beskrivelse af sprog udtrykkes i en for beskriveren naturlig form, ofte et ineffektivt arbejdsgrundlag for parseren.

Da dette forhold er et velkendt problem ved fortolkere, udformes formalismerne ofte således, at der tages mere hensyn til en effektiv fortolkning end til dens naturlighed for brugeren.

MODEL III.

Model III er modellen, hvor der nok er en præprocessor til behandling af grammatikken inden den anvendes af parseren, men hvor grammatikken altså stadig formuleres direkte i et programmeringssprog eller i en slags formalisme, der er tæt på et kendt programmeringssprog. Skematisk ser den således ud:

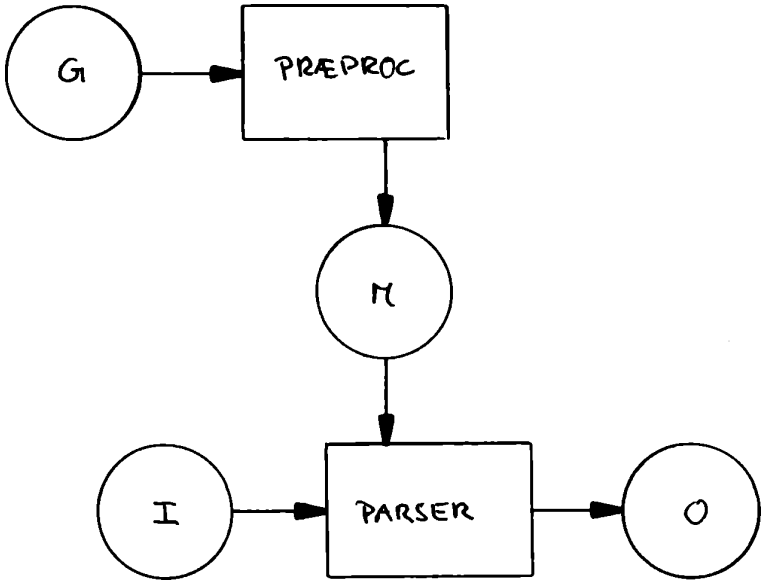


Denne model vælges ofte i systemer, hvor selve parseren skrives i et programmeringssprog, der i sig selv kan blive fortolket, fx LISP.

Præprocessorens primære opgave kan her være at undersøge, om grammatikeren har overholdt programmeringssprogets syntaktiske og semantiske regler i sin formulering af grammatikken; og måske foretager præprocessoren en mindre omformning af grammatikken. Det, der især skiller præprocessoren i denne model fra præprocessoren i næste model, er, at den her hvis den foretager en omformning af grammatikken, altid afleverer den i en programmeringssprogsform, så den direkte kan indgå i parsersystemet.

MODEL IV.

I denne model omformer præprocessoren grammatikken til en form, der af parseren anvendes som data. Denne model vælges i situationer, hvor man ønsker at være friere stillet med hensyn til grammatik-formalismens udformning, dvs. hvor man ønsker en særlig formalisme. Den kan skitseres således:



Denne udformning har sammenlignet med model II en række fordele. Her skal blot nævnes, at den ofte vil kunne give en mere effektiv parsing-proces, og at fejl opdages på et tidligere tidspunkt, og med større sikkerhed. Det sidste giver modellen et stort plus i brugervenlighed.

Præprocessorens uddata kan være på en af to former. Hvis uddata er på tabelform, kaldes præprocessoren en tabel-generator. Er uddata derimod en særlig mellemkode, der fungerer som instruktioner, kaldes præprocessoren en oversætter. I skitsen ovenfor er koden instruktioner til en maskine, der ikke eksisterer som hardware, men som simuleres af et program. En sådan maskine kaldes en virtuel maskine, og mellemkoden til den for virtuel mellemkode.

Udformningen af GESA-systemet.

I det følgende vil jeg kort opridse nogle af de overvejelser i design-fasen, der førte til, at GESA-systemet blev udformet som et oversætter-fortolker system med virtuel mellemkode.

Et af målene med GESA har været at lave et system, der gør det muligt for studerende og en en bredere kreds af lingvister at eksperimentere med lingvistiske beskrivelser og strategier.

Da der ikke kan forudsættes kendskab til programmering og da systemet primært er eksperimentelt, vil det være hensigtsmæssigt at vælge en model for systemets udformning, hvor grammatikken er adskilt fra selve parser-programmet. Model I ovenfor er hermed ude af billedet.

Det må i et eksperimental-system anses for vigtigt, at grammatikeren overfor systemet kan formulere sprogbeskrivelsen i en form som er overskuelig og gennemskuelig og samtidig føles naturlig. Det må derfor være mest hensigtsmæssigt at lave en særlig formalisme til grammatikkerne.

Parsing-systemer til naturlige sprog, vil altid være meget ressourcekrævende, dvs. processen vil kræve meget plads i maskinen og vil tage meget lang tid. Så selv om der er tale om et eksperimental-system, hvor en bruger formodentlig ikke vil prioritere effektivitet særlig højt, bør alt andet lige den mest effektive model vælges. Hermed er model II ude af billedet.

Alt i alt må model IV nok anses som den mest hensigtsmæssige udformning for et system som GESA. Til gengæld er det den vanskeligste og mest omfangsrige model at implementere. Dette kunne tale for at formalismen blev tilpasset, så systemet blev enklere, men det ville sikkert føre til det resultat, at den ikke længere opfyldte de oprindelige krav.

Det kan selvfølgelig ikke accepteres at de "lingvistiske" krav til formalismen og krav om brugervenlighed nedprioriteres på denne måde; på den anden side skal beskrivelsen jo omskrives.

Ved at betragte den formalisme sprogbeskrivelsen skrives i som et højniveau programmeringssprog og præprocessoren som en oversætter, kunne man måske komme ud over denne problemstilling.

For en ikke-datalog virkede tanken om at skulle lave en rigtig oversætter overvældende, men en nøjere gennemgang af litteratur om design af programmeringssprog og oversættere viste, at man er nået meget langt med at forene krav til programmerings-

sprog og deres oversættere med et krav om, at de sidste skulle være nemme at lave og vedligeholde.

Inden for dette forskningsområde har man især koncentreret sig om oversætterens parserdel, idet man har søgt efter metoder, der kunne klare flest af de eksisterende programmeringssprogs konstruktioner på den mest effektive måde.

Længst er man nok nået med en metode kaldet LR-parsing, der oprindelig er udviklet af Knuth (Knuth 1965) (se også Aho and Ullman 1977). Metoden kaldes LR, fordi parseren skanner inddata fra "Left-to-right" og konstruerer "a Rightmost derivation in reverse". En LR-parser har tre fordele frem for de hidtil anvendte metoder:

1. Den kan udformes, så den kan genkende en hvilken som helst sprogkonstruktion, der kan beskrives med en kontekstfri grammatik.

2. Den er mere generel end tidligere anvendte parsere og tilmed lige så effektiv.

3. Den finder syntaksfejl så hurtigt, det er muligt med strikt venstre-højre skanning af inddata.

Desværre er en LR-parser/compiler til et bestemt programmeringssprog vanskelig at implementere, hvis man starter fra bar bund. Teknikken forudsætter næsten, at man har en LR-parser/compiler-generator: dvs. at selve parseren er tabeldrevet og at der findes en præprocessor, der kan indlæse en kontekstfri grammatik og generere den tilsvarende tabel.

Har man en sådan LR-parser-generator, mangler man "kun" at lave symboltabel-behandlingen, typecheck o.lign. og kodegenerering i at have en fuldstændig oversætter. At lave en oversætter til et PASCAL-subset kan derfor gives som en 4-ugers opgave på datalogi 1. del.

LR-parsere findes i forskellige varianter. Den mest udbredte

kaldes Look-A-head LR; LALR(1)-parseren er den foretrukne, fordi den er næsten lige så effektiv som en LR-parser, men kun behøver 1/10 så store tabeller.

Implementering af en PASCAL-oversætter kan som nævnt betragtes som en overkommelig opgave, og da PASCAL og PASCAL-oversættere findes grundigt beskrevet, var der grundlag for at undersøge om de til PASCAL stillede krav kunne forenes med de krav, der var blevet stillet til formalismen i GESA.

Det viste sig at være tilfældet, og da der tilmed fandtes en LALR(1)-parser/compiler-generator, BOBS (Eriksen a.o. 1982) på vores maskine, var der kun et alvorligt problem tilbage: Hvordan skulle oversætterens uddata se ud.

Normalt er uddata fra en PASCAL-oversætter ikke en parsertabel, men derimod maskinkode, assembler eller den såkaldte P-kode. P(ASCAL)-kode er et instruktionssæt til en virtuel maskine, der som nævnt er en maskine, der ikke eksisterer som hardware. En virtuel maskine kan via fortolkning af instruktionerne simuleres af et program, der er skrevet i en eksisterende maskines sprog. Så ved at lade oversætteren generere en virtuel mellemkode, opnår man altså at gøre hele systemet mere uafhængigt af en enkelt maskine. Hvor portabelt det i sidste ende bliver, afhænger selvfølgelig også af, hvilket sprog det er skrevet i.

En anden fordel ved virtuel mellemkode er, at den kan udformes efter behov, idet instruktionssæt og maskine definerer hinanden. Instruktionssættet kan altså tilpasses, at den virtuelle maskine er en parser til naturlige sprog.

Da P-kode alene ikke hensigtsmæssigt definerer en sådan parser, var det nødvendigt at definere et nyt instruktionssæt: G-kode, hvis oversætteren skulle generere virtuel mellemkode.

Ved bestemmelsen af instruktionssættet skulle der tages hensyn til, at sættet ikke måtte blive for stort, at den enkelte instruktion ikke måtte blive for kompleks, at koden skulle

kunne genereres direkte ved første gennemlæsning af inddata (et-passage-oversætter), og endelig som nævnt, at den virtuelle maskine var en parser til naturlige sprog.

Det lykkedes at udforme et instruktionssæt, der opfyldte disse krav, og GESA-systemet blev derfor udformet som et oversætter-fortolker system med virtuel mellemkode.

Aho and Ullman 1977

A. V. Aho and J. D. Ullmann: Principles of compiler design (addison-Wesley Publishing Company 1977).

Eriksen a.o. 1982

S. H. Eriksen, B. Bæk Jensen, B. Bruun Kristensen, and O. Lehrmann Madsen: The BOBS-system, 3rd ed. (Aarhus University 1982)

Erlandsen 1983

J. Erlandsen: En introduktion til parsing og parseren GESA, in Skrifter om Anvendt og Matematisk Lingvistik 10, (Københavns Universitet 1983).

Knuth 1965

D. E. Knuth: On the translation of languages from left to right, in Information and control 8, 1965, pp. 607-639