D A T A L O G I -

L A B O R A T O R I E T

(1 9 7 4)

UPPSALA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCES

UPPSALA UNIVERSITET
INSTITUTIONEN FÖR INFORMATIONSBEHANDLING

DLU 74/A4

Februari 1974

D A T A L O G I -
L A B O R A T O R I E T

(1 9 7 4)

En beskrivning av forskningen vid Datalogilaboratoriet under 1973 och av planerna för 1974, skriven av Anders Beckman, Lennart Beckman, Mats Cedvall, Anders Haraldson, Hans-Jürgen Holstein, Sture Hägglund, Mats Nordström, Östen Oskarsson, Tore Risch, Erik Sandewall, Torgny Tholerus, Jaak Urmi och Olle Willén. Redigerad av Erik Sandewall. I forskningsarbetet har även deltagit Åke Malmberg och Klas Mårtensson.

I N N E H Å L L S F Ö R T E C K N I N G

1. Allmänt om verksamheten vid DLU*

Datalogilaboratoriet i Uppsala (DLU) är en organisation, huvudsakligen inom institutionen för informationsbehandling**, för forskning på datalogiområdet***. I DLU:s verksamhet deltar f n drygt 10 personer på hel- eller deltid.

Med denna skrift vill vi vid Datalogilaboratoriet informera om vår verksamhet. Skriften vänder sig till personer som sysslar med databehandling, men inte i första hand till andra forskare på området. Huvuddelen av skriften beskriver de projekt som pågått under 1973 och vi har då försökt beskriva bakgrunden till det arbete som gjorts; ge reflexioner och kommentarer till arbetet och skissera framtidsplanerna, ange vad som blivit gjort. Däremot har vi undvikit att gå in på detaljer t ex i programbeskrivningar; istället hänvisas till de forskningsrapporter som anges i bilaga 2. Vissa arbeten som bedöms vara av särskilt intresse, presenteras dock i detalj. En mindre, avslutande del av skriften beskriver föreslagen, ny verksamhet för 1974/75.

---

* Detta kapitel har översiktskaraktär, och är ett något reviderat utdrag ur motsvarande kapitel i föregående årsrapport (DLU -72).

** DLU består av två forskningsgrupper, för datalogi och för sociocybernetik. Den förra ingår i institutionen för informationsbehandling, den senare dels där, dels i sociologiska institutionen.

*** Med datalogi avser vi ett studium av datastrukturer, programmeringsspråk, och andra "mjukvaru"-hjälpmedel vid användning av datorer. Till datalogi för vi: programmeringsspråk, översättarteknik, datastrukturer, operativsystem, "theory of computation", artificiell intelligens, formelmanipulation. En mer detaljerad definition gavs i föregående årsrapport (DLU -73).

I detta inledande avsnitt skall vi på ett fåtal sidor först beskriva målsättningen och uppläggningen av arbetet under 1973. Målsättning och uppläggning är i stort sett identisk med i föregående årsrapport. (DLU -73).

Det har varit vår målsättning att denna årsrapport skall kunna läsas oberoende av de föregående. Därför är kapitel 1 och 2 av översiktskaraktär och i stor utsträckning omtryck av (delar av) motsvarande kapitel i föregående årsrapport (DLU -73). Övriga kapitel är helt nyskrivna.
Definition av forskningsområdet. Verksamheten vid DLU inriktas f.n. på metodproblem vid intensiv bearbetning av små databaser, varvid bl.a. resultat från forskning i artificiell intelligens kommer till användning. Låt oss närmast specificera begreppet liten databas.

Vi konstaterar först att så gott som varje datorprogram arbetar med en viss mängd data, som beskriver objekt eller företeelser i omvärlden. I vissa fall (t.ex. simuleringar, många beräkningsprogram) existerar denna datamängd bara medan programmet exekveras, eftersom den genereras av programmet, och inte sparas efter körningen. I många andra fall (t.ex. de flesta administrativa tillämpningar) har data en större livslängd: en viss datamängd bearbetas tid efter annan av ett eller flera program. Om en sådan datamängd uppfyller vissa villkor på systematisk uppbyggnad, applikationsoberoende, m.m. kallar vi den en databas.

Databasen tillordnar normalt egenskaper åt objekt; därvid är objekt och egenskaper (eller åtminstone många av dem) givna redan genom problemets specifikation. Exempel: objekten kan vara personer; egenskaperna kan vara personens adress, födelsenummer, årsinkomst, osv.

Det är välkänt att egenskaperna kan vara strukturerade; t.ex. kan man för en viss person, för flera successiva år, vilja ange en persons inkomst, mantalsskrivningskommun, m.m. Nederst i denna struktur har man emellertid vissa element, vilka kan vara

en teckensträng

en numerisk storhet (på vilken aritmetiska operationer kan utföras)

men också

en referens till ett annat objekt

Enklaste exempel: om två personer, man och hustru, båda är representerade i databasen, kan man vilja referera från (representation för) maken till (representationen för) hustrun.

Databaser där referenser mellan objekt förekommer, kan vi kalla referensdatabaser.* Vid arbete med sådana databaser uppkommer ett antal intressanta problemställningar, som inte finns för referensfria databaser. Struktursökning i databasen blir oftast önskvärd, dvs man vill utgå från ett objekt och följa refernser till relaterade objekt ett eller flera steg, för att uppsöka information som är relevant för det givna objektet.** Denna struktursökning kan ibland formuleras som en slutsatsdragning. Inmatning och utmatning i databasen, eller med andra ord människa-maskin-kontakten, kan lösas ganska enkelt för referensfria databaser, men får många nya aspekter när referenser kan ingå. Nya krav uppstår på det använda programmeringsspråket. Dels finns det omedelbara kravet att man skall kunna manipulera med referenser, men det finns även subtilare konsekvenser, såsom

---

*Ej att förväxla med relationsdatabaser, vilka kan men inte behöver innehålla referenser

**Observera att sådan struktursökning inte har något väsentligt gemensamt med den uppsökning som förekommer i konventionella databaser, när man vill uppsöka ett objekt med viss given egenskap, och därvid använder t ex binärsökning. Vid vissa implementationer av referensdatabasen (jfr nedan) kan dock varje steg i struktursökningen kräva en uppsökning.

att man i programmet skall kunna omnämna inte bara konstanta numeriska storheter (t ex 144) och konstanta strängar (t ex "GRODA") utan även konstanta referenser eller "noder" i databasen.

Ovanstående problem uppkommer genom förekomsten av referenser, och kan betecknas som principiella. Därutöver finns i praktiskt programmeringsarbete ett antal organisatoriska problem, som ofta direkt eller indirekt har att göra med minnesstruktur (uppdelning på primär- och sekundärminne): hänsyn måste tas till minnesstorleken, t ex genom att man väljer en "smart" packning av data; hänsyn måste tas till överföringshastigheter mellan olika minnesnivåer, och de olika faktorer som påverkar denna, och en lämplig dataorganisation måste väljas därefter, osv.

Dessa organisatoriska problem bortfaller om man har tillräckligt liten databas eller tillräckligt stort primärminne, så att hela databasen kan läggas ut där med en standardiserad representation. De kan också bortfalla (eller åtminstone göras osynliga för programmeraren) om databasen är måttligt mycket större än primärminnet, och man använder tekniken med virtuellt minne.

I arbetet vid Datalogilaboratoriet har vi valt att utesluta dessa organisatoriska problem, och koncentrera oss på dem som ovan kallats principiella. Det har skett bl a genom att vi byggt och använt programmeringssystem som har programmerat virtuellt minne. Skälen för detta val är flera: lösningar på de organisatoriska problemen tycks bli hårt bundna till resp. tillämpning, medan de principiella problemen tillåter mer allmängiltiga lösningar; vi kan studera de principiella problemen utan att gå in på de organisatoriska, men knappast vice versa, och vi måste begränsa oss; och inte minst: genom minnesteknikens utveckling tycks de organisatoriska problemen om inte försvinna så i varje fall kraftigt förändras. En referensdatabas som studeras map. principiella men inte map. organisatoriska problem, betecknar vi nu som en liten databas. Det är studium och metodutveckling för dessa som är den röda tråden i forskningen vid Datalogilaboratoriet.

Avslutningsvis kan en kommentar beträffande smådatabasers förhållande till simuleringar vara på plats. Vid händelsestyrd simulering har man en datamängd som är rik på referenser, vilket avspeglar sig i förekommande programmeringsspråk för sådan simulering. Å andra sidan är den simulerade situationen normalt bara representerad under själva körningen, dvs man har ingen databas i ovanstående betydelse. Detta leder till att problem med människa-maskinkontakt inte är aktuella. Förekammande struktursökning i simuleringsprogrammets datastruktur tycks också vara rätt problemfri. Av dessa skäl faller konventionell händelsestyrd simulering inte inom vårt problemområde små databaser. Det kan dock finnas mindre konventionella fall av simulering, där smådatabasproblemen återkommer, t ex interaktiv simulering, och simulering där man vill ha noggranna protokoll av enstaka händelser i det simulerade förloppet.

Strategi vid DLU för forskning på smådatabasteknik. En första princip är att bedriva arbete på metoder och programredskap parallellt med arbete på experimentella tillämpningar av de utvecklade metoderna och redskapen. Experimenten är nödvändiga för att hålla metodutvecklingen på rätt kurs och hindra att den får för stor teoretisk slagsida.

En andra princip är att programmeringssystem och program skall göras starkt interaktiva. Detta motiveras av att så gott som alla smådatabastillämpningar kräver stark interaktion mellan användaren och maskinen. En uppdelning på delproblem inom smådatabasområdet kan sedan göras på flera olika sätt. Man kan skilja på data-arkitektur (strukturering av data) och program-arkitektur. Vi kan också skilja på interna problem i programmet, och interaktionsproblem med användaren. Kombineras dessa två dimensioner får vi följande uppställning:

	data-arkitektur	Programarkitektur
internt	hur organisera data från en given tillämpning som en lämplig datastruktur?	hur göra de önskvärda sökningarna o a manipulationerna i databasen?
interaktion	hur välja en lämplig notation för kommunikation med systemet? hur ställa upp delar av databasen i tabellform för presentation?	hur översätta mellan extern och intern notation?

Dessutom har vi en tredje dimension som är ortogonal mot de två ovanstående, nämligen komplexitets graden. Vid låg komplexitet har vi det extrema fall där varje objekt har ett litet antal fixa indikatorer, och egenskapen under varje indikator är en sträng, ett talvärde, eller en referens. Här kan samtliga fyra problem i ovanstående uppställning få enkla lösningar. Vid mycket hög komplexitet har vi (som gränsfall) det fall där den externa notationen är naturligt språk, t ex svenska, och där man alltså internt måste kunna representera och manipulera sådan information som uttrycks i naturligt språk. Vi tror att man i de flesta praktiska problem rör sig mellan dessa två nivåer: att det första fallet inte räcker till och det senare inte behövs. Detta leder till önskemål om en formelrepresentation (hierarkisk representation) i dataarkitekturen.

På avdelningen programarkitektur är förstås den första frågan: hur skall programmet för en viss smådatabastillämpning vara utformat? Vi kan se följande möjligheter:

(a) använd ett generellt program, som kan anpassas till olika tillämpningar

(b) använd ett programmeringsspråk, som är anpassat för problemklassen smådatabastillämpningar, och i vilket man kan skriva ett program för den aktuella tillämpningen. - Det finns då några del-möjligheter:

(b1) nyskriv hela programmet för denna tillämpning

(b2) använd i programmet subrutiner som valts från ett programbibliotek. Man får inte förvänta att varje subrutin skall vara användbar i varje tillämpning, utan istället välja rätt rutin för varje problem. Rutinen måste förstås förses med parametrar som karakteriserar den aktuella tillämpningen.

(b3) använd i programmet subrutiner som genererats för att passa den aktuella tillämpningen. Möjliggör åtminstone i princip effektivare program än (b2), men liknar eljest detta fall.

Om man väljer alternativ (a) kan man skriva det generella programmet i ett lågnivåspråk ss. assembler. Om man gör hårda begränsningar på

tillåtna datastrukturer o dyl kan man då få hög effektivitet för de problem som tål begränsningarna. Om man emellertid gör programmet enligt (a) så generellt att det klarar många praktiska tillämpningar, får man dels effektivitetsproblem (vid varje körning får man också betala för alla faciliteter som man inte använder), dels administrativa problem med att hålla ihop arbetet på ett mycket stort program. Av dessa skäl har vi vid DLU konsekvent valt alternativ (b), dvs användning av ett för smådatabaser lämpat programmeringsspråk. Beträffande alternativ (b1), (b2) och (b3) är det däremot svårt att generellt ange något som bättre än de andra. Arbetet bedrivs därför så att vi får pröva och skaffa erfarenhet från alla tre angreppssätten. Metod (b3) är principiellt intressant och förefaller lovande, och en väsentlig del av arbetet läggs ned på denna, men även övriga metoder används. Vid användning av metod (b2) och (b3) är det nödvändigt att välja ett programmeringsspråk som substrat för programbiblioteket resp. programgeneratorerna, och att hålla fast vid detta. Av skäl som redovisades i en tidigare årsrapport (DLU-72) har vi valt LISP som sådant språk, Datorsituationen har nödvändiggjort att systemprogrammering, främst i form av implementering av LISP-system, tagits upp som en gren av verksamheten.

Tillkommande forskningsområden. Datalogi är ju inte bara smådatabaser, och vi ser det som önskvärt att verksamheten vid DLU i rimlig takt utvidgas till andra delar av datalogin. Några exempel på detta ges i denna årsrapport. Arbetet på systemprogrammering har givits naturliga anknytningar till studium av implementeringsspråk (se kapitel 3). Arbetet på programgeneratorer (metod b3 ovan) anknyter naturligt till det programmanipulerande programmet FFC, som manipulerar Fortran och även är skrivet i Fortran. FFC beskrivs i kapitel 5. Under hösten bedrevs arbete på "Totala programmeringssystem, särskilt inkörningshjälpmedel" inom ramen för projekt P, och med inriktning på konventionella programmeringsspråk. Planer för arbetet beskrevs i föregående årsrapport. I detta arbete fanns många möjligheter att ta idéer från våra LISP-system: ett LISP-system av INTERLISP-typ är ett totalt programmeringssystem, som programmeraren arbetar med under programutvecklingen, och som innehåller ett stort antal inkörningshjälpmedel. -- Slutredogörelsen för denna arbete är ännu ej utformad, varför arbetet ej beskrivs i denna årsrapport. Vi hoppas att det skall vara möjligt att fortsätta dessa utvecklingslinjer, och t ex tillämpa metoder för

manipulation av LISP-program även i andra språk. I en sådan utveckling bidrar den formella semantiska definitionen av Algol och Simula som beskrivs i kapitel 6a.

Former för verksamheten under 1973. Arbetsfördelningen under året, dvs uppgift om vilken eller vilka personer inom DLU som utfört arbetet på projekt anges vanligen ej i redogörelsen för resp projekt, utan har samlats till en översikt i bilaga 1.

Datalogilaboratoriet har under året varit representerat vid följande konferenser:

SIGPLAN/SIGMICRO Interfåce Meeting (Urmi);
Third International Joint Conference on Artificial Intelligence (Stanford) (Haraldson, Sandewall, Urmi);
ACM Symposium on Principles of Programming Languages (Nordström);
Symposium "Organischmische Informationsverarbeitung" (Östberlin) (Sandewall);
Symposium "Collective Behaviour of Automata" (Leningrad) (Sandewall).

Under året har Datalogilaboratoriet gästats av följande föreläsare:

Dr Daniel Bobrow (Xerox Palo Alto Research Center)
Professor Thomas Bredt (Stanford University)
Professor Donald Knuth (Stanford University & Oslo universitet)
Professor Roger Schank (Stanford University och Lugano)
Dr John Walters (IBM Palo Alto)

Verksamheten vid Datalogilaboratoriet under 1973 har möjliggjorts genom ett generöst bistånd från ett flertal håll. Den har finansierats genom bidrag eller uppdrag från Försvarets forskningsanstalt (Planeringsavdelningen), IBM Svenska AB, Naturvetenskapliga forskningsrådet, Riksbankens Jubileumsfond, samt Styrelsen för teknisk utveckling. Inom Uppsala universitet har vi som en del av institutionen för informationsbehandling på olika sätt tillgång till dess resurser, och Uppsala Datacentral har finansierat utvecklingen av Interlisp/370. Slutligen har vi för verksamheten även disponerat stud-medel för körningar på UDAC och QZ.

Arbetet har stimulerats genom den goda arbetsmiljön och det breda verksamhetsfältet vid institutionen för informationsbehandling, samt genom den nära kontakten med Uppsala Datacentral och dess mångfacetterade verksamhet. Under året har vi haft glädje av stimulerande kontakter, diskussioner och samarbete med motsvarande grupper vid följande institutioner:

Datasaab
FOA
IBM Svenska Aktiebolag
KTH (Institutionen för informationsbehandling)
Statistiska centralbyrån (enheten för databehandlingsmetodik)
Stockholms universitet (CADIS-projektet)

Till alla dem som på olika sätt bidragit till verksamheten vill vi här framföra ett tack.

P R O G R A M M E R I N G S S Y S T E M

2. Bakgrund och översikt*

Med ett programmeringssystem menar vi den samling av program som understöder programmerarens arbete i ett visst programmeringsspråk. Här ingår alltså kompilator och/eller interpretator, editeringsprogram, felsökningshjälpmedel, osv.

2a. Principer för arbetet på programmeringssystem vid DLU

De flesta datoranvändare utnyttjar förefintliga programmeringssystem. Vid Datalogilaboratoriet har vi emellertid utvecklat egna sådana, av två skäl:

(a) smådatabaserna ställde särskilda krav som inte uppfylldes av förekommande system

(b) intresse för att få igång forskning på området programmeringssystem

Huvuddelen av arbetet har satsats på att bygga system för programmeringsspråket LISP. Valet av detta språk motiverades utförligt i en föregående årsrapport (DLU-72). LISP som språk har funnits sedan lång tid tillbaka, och vi har alltså endast gjort nya system på en maskin där det inte förut var implementerat (Siemens 305) och en där vi inte var nöjda med existerande implementation (IBM 360), samt ett portabelt system skrivet i Fortran, kallat LISP F1 (reviderat d.å.). Vidare göres arbete på olika "överbyggnader" av LISP-systemet, samt grundläggande arbete på (maskinnära) implementeringsspråk.

---

* Detta kapitel har översiktskaraktär, och är ett något reviderat utdrag ur motsvarande kapitel i föregående årsrapport (DLU -73).

LISP:s principer för uppbygnad av programmeringssystem är mycket intressanta, men genom den (åtminstone vid första påseende) obekväma notationen i språket LISP torde detta knappast komma att nå någon större spridning. Vi försöker ändå på två sätt föra ut resultat från vårt arbete på LISP-system:

- förbättra LISP:s notation, så att den blir bekvämare speciellt för nybörjaren. Den resulterande notationen ytspråket kallas PLAST och beskrivs i kapitel 4f.
- överföra den teknik som visat sig nyttig i LISP till andra programmeringsspråk i den utsträckning det går. En grund har lagts genom programmet FFC (se kapitel 5) som behandlar Fortran. FFC utvecklades dock i första hand med sikte på en näraliggande tillämpning vid Uppsala datacentral. Planerna på en fortsättning beskrivs i kapitel 5c.

Utöver PLAST har också annat arbete bedrivits på vidareutveckling av programmeringsspråk. Här ingår dels en formell definition av semantiken i Algol 60 och Simula 67 (ett i princip helt teoretiskt arbete, som vi dock tror kommer att få vissa tillämpningar), dels studier av hur en ev. framtida efterträdare till LISP borde se ut. Det senare har resulterat i språket REC (se kapitel 4g). Det skall dock samtidigt påpekas att all programmering vid DLU görs i LISP (utom FFC som gjordes i Fortran), och att övergång till att använda en efterträdare till LISP troligen ligger långt fram i tiden. Slutligen pågår arbeten på programmeringssystem för andra typer av databehandling än smådatabaser och för t.ex. Fortran; se föreg. årsrapport samt kap. 5 i denna rapport.

2b. Struktur hos programmeringssystem för LISP

En mycket väsentlig egenskap hos LISP är att det är ovanligt lätt (jämfört med andra språk) att skriva programgenererande program, dvs program som från en mer tillämpningsorienterad specifikation av en uppgift genererar ett LISP-program som utför uppgiften. Till skillnad från de flesta andra språk (utom maskinkod) är det

också möjligt att göra detta dynamiskt, dvs att generera och exekvera kod i samma jobbsteg. Denna möjlighet har utnyttjats flitigt vid arbetet inom DLU. En sådan programgenerator kan uppfattas som en kompilator från ett "högre" språk till LISP (varifrån det kan vidare kompileras till maskinkod). Genom detta förhållande kan begreppet "programmeringssystem" tolkas mycket brett. Vi kan urskilja följande typer av moduler i vårt totala programmeringssystem:

a LISP-systemet, maskinkodsorienterad del.
Detta inkluderar den kod som skrivits i assembler (i princip interpretatorn och de rutiner den anropar), samt den del som skrivits i LISP men som generar en maskinkodsliknande struktur (dvs kompilatorn)

b LISP-systemet, LISP-kodad del.
Detta inkluderar struktureditor, felsökningshjälpmedel osv som skrivits i LISP, och som inte tar någon hänsyn till maskinkoden, utan "skrivits för en LISP-maskin".

c Ytspråkssystem
Detta är system som översätter från ett mer konventionellt (t.ex. algol-liknande) programmeringsspråk till LISP.

d Programoptimerare
En viktig teknik för att skriva programgeneratorer i LISP är att först skriva en enkel generator som genererar jämförelsevis ineffektiva LISP-program, och sedan skicka dess resultat till ett generellt optimeringsprogram. I det enklaste fallet kan generatorn vara: "tag följande generella, parameterstyrda program och sätt in följande värden på parametrarna", dvs helt trivial. Programoptimeraren gör då nästan hela arbetet.

e Programgeneratorer
Genererar program enligt givna specifikationer, inom ett visst problemområde.

Pga skillnader i programmeringsteknik mm har vi dock föredragit att göra en uppdelning så att endast a-c kallas programmeringssystem och behandlas i kapitel 2 - 6 av denna rapport, medan d-e räknas till programhjälpmedel. Programoptimerare behandlas i kapitel 8, programgeneratorer ingår i kapitel 9 och 10.

3. Maskinnära system

Under året har det uppkommit två arbetsuppgifter som ligger "under" (mer maskinnära än) programmeringssystemet, LISP eller motsvarande. Här ingår dels en studie av några konkreta implementeringsspråk samt av problemställningar kring implementeringsspråk i allmänhet, som gjordes inom projekt P, dels arbete på en korsassembler. Båda dessa uppgifter utfördes som "utlöpare" av den mer ordinarie verksamheten.

3a. Studium av implementeringsspråk

Med implementeringsspråk förstås vanligen ett högrenivåspråk avsett att användas vid implementering av programvara (programmeringssystem, operativsystem etc). För denna typ av programmering har tidigare assembler varit det vanligaste (och oftast enda tillgängliga) språket. Detta har haft flera orsaker, bl a att bara assembler tillåtit fullständig kontroll över adresseringsmekanismer (för avancerade datastrukturer) och andra mer hårdvaruberoende funktioner som bitmanipulering och interrupthantering.

Under senare år har det utvecklats flera nya programmeringsspråk avsedda att ersätta assembler. Dessa kan grovt sett indelas i två kategorier, MOL och SIL.

Ett MOL (Machine Oriented Language) är ett maskinorienterat språk av högrenivåtyp som är avsett att helt ersätta assembler på en given maskin.

Ett SIL (Systems Implementation Language) däremot är ett språk avsett för design och implementering av systemprogramvara. Det innehåller därför oftast möjligheter till komplicerade datastrukturer, en rik flora av kontrollstrukturer och alla de datatyper man vanligen behöver i sådana system. Ett SIL är vanligen ganska maskinoberoende.

Ett naturligt ömskemål är att MOL och SIL för en maskin skall vara samma språk. På äldre maskinvara är det ofta svårt att få ett SIL som är problemnära och som samtidigt kan kompileras till effektiv kod utan en gigantisk optimerande kompilator. Har man däremot möjlighet att själv (t ex

med mikrokodning) bestämma maskinens instruktionsuppsättning är det (i varje fall teoretiskt sett) betydligt enklare att göra ett SIL som ger förhållandevis effektiv objektkod.

Detta var i början av 1973 situationen för Datasaab, som hade en mikroprogrammerbar centralenhet (FCPU), och som inom projekt P behövde ett MOL/SIL. Eftersom centralenheten är mycket flexibel, var det mycket attraktivt för Datasaab att välja ett existerande SIL och (ev efter smärre modifikationer) använda detta som MOL/SIL utan att effektiviteten därför behövde bli låg.

Inom projekt P gjordes därför utvärderingar av existerande implementeringsspråk. Vid DLU studerades bl a Bliss och CORAL-66. Båda dessa är förhållandevis maskinnära (mera MOL än SIL) och sållades så småningom bort. Valet föll på MARY, ett Algol-68-inspirerat språk utvecklat i Trondheim av bl a Mark Rain och egentligen avsett för en serie minidatorer.

Det är intressant att studera de faktorer som avgjorde valet, eftersom de kan tjäna som riktlinjer för vidare arbete.

En av de viktigaste faktorerna var säkerheten. Bliss tillåter ingen typdeklarering av variabler, medan MARY har boligatorisk typdeklarering och typkontroll (även av pekare) under kompileringen. I MARY finns dessutom hjälpmedel för att förhindra överskridandet av arraygränser, förutom väljbar dynamisk kontroll även sådana satskonstruktioner där kompilatorn lägger in aktuella gränser, tagna ur arraydeklarationen. Ingen av dessa möjligheter finns i Bliss, som i alla lägen tillåter alla möjliga bitsträngar som adresser.

I fråga om kontrollstrukturer var Bliss och MARY tämligen likvärda. I båda språken ingår de kontrollstrukturer man i dagsläget anser väsentliga: if..then..else, case-konstruktioner, villkorliga loopar (do until etc) och for-loopar. I Bliss har den traditionella goto-satsen helt borttagits. Som komplement till de ovan nämnda kontrollstrukturerna finns

i Bliss i stället leave-satser, med vars hjälp man kan lämna en omgivning som man befinner sig i (t ex ett block, en loop etc). MARY har kvar goto-satsen men med sådana inskränkningar att den i praktiken i mycket liknar leave-satsen i Bliss.

I MARY kan man på ett tämligen normalt sätt definiera datastrukturer. I Bliss däremot specificerar man åtkomstalgoritmen för elementen i datastrukturen. Samma åtkomstalgoritm kan sedan lätt "avbildas" på flera olika fält, och åtkomstalgoritmen för ett element i en struktur kan lätt ändras utan att elementreferenserna behöver ändras. Även om många anser att datastrukturdefinitionerna i MARY kunde förbättras medför de mycket större säkerhet och bättre läsbarhet än åtkomstalgoritmbeskrivningarna i Bliss.

Bliss är mycket väl anpassat till den dator det är utvecklat för, PDP-10 (DEC-10). Detta medför att Bliss är mest lämpat för en dator med PDP-10-liknande arkitektur. För att få ett effektivt MOL/SIL hade man alltså vid val av Bliss fått göra P-maskinen till en PDP-10-liknande maskin. Detta är ett av huvudproblemen vid val av MOL/SIL till en ny maskin - väljer man ett existerande språk låter man samtidigt språkkonstruktören få ett avgörande inflytande på maskinens arkitektur, något som denne kanske inte alls tänkt sig (han kanske bara gjort bästa möjliga språk till en existerande, ibland ganska risig maskin).

MARY är avsett för en familj av minidatorer, men är gjort för att vara anpassningsbart till de flesta (konventionella) maskiner. Det finns i MARY inga så uppenbart "en-maskinorienterade" detaljer som i Bliss. I gengäld medför detta att MARY inte utan en mycket optimerande kompilator kan bli effektivt på en maskin med "specialiserade och finessrika" instruktioner. Därmed inte sagt att MARY skulle ge dålig kod på en maskin med normal instruktionsuppsättning.

Eftersom det var aktuellt att ta ett existerande språk kom helt naturligt kompilatorerna för dessa språk att tilldra sig intresse. Blisskompilatorn är mycket stor och gör mycken optimering för att kunna ge

en kod som är jämförbar i effektivitet med assemblykod. Detta är ett resultat av att man vid bedömning av MOL och delvis även SIL gärna gör jämförelser i effektivitet på mindre rutiner kodade i språket och i assembler snarare än jämförelser av "global effektivitet", läsbarhet, underhållskostnader, felfrekvens etc. Just det faktum att den existerande Bliss-kompilatorn var så stor blev ett argument mot språket Bliss ("går inte att kompilera på små system"etc).

Diskussionerna om MOL/SIL till projekt P fördes till stor del vid möten på Datasaab. En del av de synpunkter som framfördes av personal från DLU finns redovisade i rapporter: Bliss är beskrivet i DLU 73/A6, Coral-66 i DLU 73/A9 och i DLU 73/A8 framförs några allmänna synpunkter på implementeringsspråk.

3b. Korsassembler för Nova-maskinen

Under 1973 bedrevs arbete på NOVALISP, en korskompilator för LISP från IBM 360 till en Nova-minidator. Detta arbete beskrivs i avsnitt 4c. Det var först meningen att korskompilatorn skulle generera assemblerkod för Novan på papersremsa, varefter denna skulle assembleras med Novans assembler. Detta visade sig emellertid opraktiskt med hänsyn till den tillgängliga minidator-konfigurationen (snabb remsläsare saknades), assemblerfaciliteterna, m m. Därför skrevs istället ett LISP-program som utför assembleringen på 360-maskinen, vilken alltså får utföra allt arbete ända fram till objektkoden för Nova. LISP valdes som programmeringsspråk eftersom målsättningen var att så snart som möjligt få tillgång till en fungerande korsassemblator, som lätt skulle kunna byggas ut för att ingå som del i annat program.

Det fanns ingen möjlighet att överföra resultatet av assembleringen från 370 till Novan på annat sätt än på lista, varför det blev naturligt att inte assemblera och testa större rutiner än att det blev överkommligt att med hjälp av teletypen lägga in den oktala koden i Novan. Assemblatorn har därför två argument som input: assembly koden och en symboltabell varav symboltabellen får utelämnas. Godtyckliga delar av programmet kan assembleras oberoende av varandra genom att externa referenser tilldelas värden i symboltabellen.

Novan har endast helordsinstruktioner vilket medför att endast symbol och loccount:s värde måste sparas i symboltabellen. Syntaxen av assemblykoden skiljer sig något från Nova-assemblatorns men ett tillägg till korsassemblatorn kan lätt göras om icke "LISP-tillvända" användare störs av formatet. Korsassemblatorn tillåter Macros, numeriska lablar, adresskonstanter, ändring av gamla eller införande av nya operatorer. Operatorerna indelas i två huvudtyper, som i fortsättningen kallas typ A och typ B. Alla typ A operatorer skall finnas före första operator av typ B. Till typ A hör macrodefinitioner och de operatorer som används för att skapa nya eller ändra gamla operatorer av typ B. Typ B operatorna består av assemblatorinstruktioner, kommentarer, macroanrop, minne-register operatorer, minne-minne operatorer, aritmetiska och logiska operatorer, in och utmatning med register-argument, och slutligen in och utmatning.

Indelning i typer har gjorts för att assemblatorn skall kunna testa om argumenten uppfyller operandens krav och om så ej är fallet ge felmeddelande. Assemblatorn testar operatorerna, i den ordning de står uppräknade ovan, på vilken typ de hör vilket medför att om en operator definierats i flera typer kommer den som ligger först att användas.

Definition av ny operator: En operator av typ A som anger till vilken undertyp den nya operatorn skall bindas samt två argument: den nya operatorns namn och oktala värde. Ex. (xALCx NYOP 37001) tilldelar den nya operatorn NYOP undertypen ALC och värdet 37001 oktalt.

Omdefinition av gammal operator: För att undvika misstag p g a ordningen vid genomsökningen av de olika typerna definieras en macro med den gamla operatorns namn. Macron skall vid expansionen ge en ny operator definierad enligt ovan. Macron kommer då att, på grund av genomsökningsordningen upptäckas före den ursprungliga definitionen av operatorn.

Macroexpandering: Den funktion som associeras med macrons namn evalueras med macroanropets operander som lambda-variabler. Resultatet av evalueringen läggs in i stället för macroanropet.

Status: Korsassemblatorn som är grundligt testad finns upplagd på bibliotek tillgänglig för intresserade Nova-programmerare.

4. Programmeringssystem för smådatabaser

Det primära programmeringsspråket vid Datalogilaboriet är LISP, vilket ger en gemensam grund för verksamheten. Härigenom har också implementering av LISP-system kommit att ingå i arbetet. Under 1973 har arbetet på ett system i dialekten INTERLISP för IBM 360/370 fortsatt och nästan slutförts. Systemet har i olika pre-releaser använts av olika personer vid DLU under större delen av 1973 och visat sig vara mycket pålitligt och lätt att arbeta med. Samtidigt har vårt portabla LISP-system, LISP F1, dels använts som material för experiment med effektivitetsmätning, dels något vidareutvecklats.

Samtidigt vill vi bevaka olika alternativ till och möjliga vidareutvecklingar av det använda programmeringsspråket. Avsikten är att när tillräckligt många, väsentliga, och väl genomtänkta förslag till förändringar har ackumulerats, skall ett byte av språk kunna ske. Härvid bevakar vi naturligtvis främst litteraturan på området, men visst arbete görs även här. Under året har arbetet fortsatt på programmeringsspråket REC, vilket har många grundläggande principer (ss läs- och skrivbara datastrukturer, atombegrepp, och representation av program som datastrukturer i språket) gemensamma med LISP, men som samtidigt skiljer sig från LISP bl a genom en renare struktur och en annorlunda typ av parameterkommunikation.

För det kommande året föreslås också ett studium av den nya generationen av mycket-hög-nivå-språk som framkommit inom artificiell-intelligensforskningen. Detta motiveras närmare i kapitel 16.

4a. Uppsala-BBN-LISP eller INTERLISP/360

Arbetet med implementation av Uppsala-BBN-LISP har fortsatt under året och är nu praktiskt taget avslutat. Återstående arbete, som huvudsakligen består av dokumentation, beräknas avslutas under de första månaderna under 1974. Under året har användare vid Uppsala Datacentral och DLU haft tillgång till en version som har fungerat problemfritt. På denna version har större delar av de vid DLU utvecklade program enheten implementerats. En intressant funktion hos systemet som förtjänar uppmärksamhet, är "SAVE"-funktionen som möjliggör att det virtuella minnet sparas på yttre minne på sådant sätt att systemet kan återstartas från detta.

Systemet består av en assemblykodad del och en LISP-kodad del.

Systemet initialiseras genom att den assemblykodad delen utgör basen i följande bootstrapping process.

(a) Grundsystemet (endast assembly-kodade delen) får läsa in filer med LISP-kodade definitioner av resten av systemet inkluderande bl a

grundläggande lispfunktioner
editor
debuggning hjälpmedel
rutiner för formulerad utmatning
lispkompilator och assemblator
vissa omdefinieringar av assemblykodade systemfunktioner

(b) Den i steg (a) inlästa lispkodade delen av systemet kompileras med hjälp av kompilatorn, (som körs interpretativt tills den kompilerat sig själv), som en effekt av detta skapas filer innehållande kompilerad kod (objektkod) för hela den lispkodade delen av system.

(c) Grundsystem (endast assemblykodade delen) får läsa in den i steg (b) sparade kompilerade koden och därmed ge alla funktioner i systemet en definition i maskinkod. Vi har nu i det virtuella minnet i princip ett fullständigt system bestående av runtimesystem, interpretator och en mängd funktioner i maskinkod, som antingen fåtts genom att skrivas för hand (assemblykodad del) eller genom att kompilera LISP kod. Hittills har all paging skett på ett enda pagingdataset med namnet "SWAPDS".

(d) Vi exekverar nu funktionen SYSSAVE som finns definierad i den assemblykodade delen, varvid följande skrivs ut på ett dataset "SYSSAVE"

(i) alla aktiva pagear skrivs. Med aktiv page menas en page som innehåller information, som beskriver systemet. Hit hör inte allokerade men ännu icke utnyttjade pagear.

(ii) Pagetabellen

(iii) delar av datararean

(iv) delar av assemblykodade delen.

Stackar o d, skrivs icke ut.

Uppsala-BBN-LISPen definieras nu av en laddmodul samt detta dataset

"SYSSAVE". När en användare ska använda systemet allokeras två dataset för detta, ett "slaskdataset" som är det tidigare nämnda SWAPDS och SYSSAVE som definierar systemet. SYSSAVE delas mellan alla användare och är endast tillåtet för läsning. Stackareor och nya areor i användarens virtuella minne allokeras på SWAPDS: Om användaren försöker ändra en page på SYSSAVE upptäcks detta av pagingrutinerna och användaren får en privat kopia av den aktuella pagen till SWAPDS. Detta innebär att pagear princip kommer att "vandra" från SYSSAVE (systemet) till SWAPDS (användarens privata area) om användaren definierar om systemet eller delar därav. Användaren har också möjlighet att spara sitt virtuella minne för senare fortsatt exekvering. Detta görs med funktionen SAVE. När SAVE exekveras kommer en användares alla aktiva pagear som inte finns i systemet (dvs användaren har skapat dem) och pagear som definierats om att sparas i ett dataset "USERSAVE". "USERSAVE" kan alltså sägas vara skillnaden mellan systemet som det är definierat av SYSSAVE och en användares utvidgade och ev. omdefinierade system. Användaren kan sedan starta upp ett system från "SYSSAVE" och "USERSAVE" som ser ut enligt hans egna önskemål.

SAMMANFATTNINGSVIS

Följande dataset används eller kan användas för paging och uppstartning samt sparande av virtuellt minne.

namn	läsning/ skrivning	används till
SWAPDS	läs/skriv	som paging dataset under en session. sparas aldrig
SYSSAVE	läs	läsning av pagear som definierar systemet
USERSAVE	läs	läsning av pagear som definierar de ändringar en användare gjort till systemet.

Nya SYSSAVE och USERSAVE kan skapas genom funktionerna (SYSSAVE) och (USERSAVE). Fördelen med det ovan beskrivna arrangemanget är:

1. alla användare delar på ett dataset som definierar systemet dvs "SYSSAVE". Detta förväntas bli 50-100 spår på en IBM 3330-skiva (varje spår ca 13 Kbytes).

2. En användare behöver för att spara ett system bara ha ett dataset själv som exekt tymmer ändringarna. Dvs USERSAVE medför att vi inte har dubbellagring av information.
3. Temporära areor (SWAPDS) behöver bara finnas allokerade medan jobbet är aktivt.

Ovanstående minskar väsentligt behovet av externa minnesareor för att spara olika användares olika versioner av systemet.

4b. Studium av hashningsmetoder

I ett LISP-system har man en typ av records, atomer, som står i ett-ett-motsvarighet, till teckensträngar. Varje gång en atom skall skrivas ut (i en print-sats el dyl) skriver man ut den associerade teckensträngen, och varje gång systemets allmänna read-rutin läser denna teckensträng, går den igenom en symboltabell och slår upp motsvarande atom, så att identiskt samma record (samma minnesadress återfinnes. I atomen kan sedan ligga referenser till andra datastrukturer som karakteriserar atomer (t ex egenskapslistor).

Vid all inläsning av data, och även av program, går man alltså igenom symboltabellen. Av effektivitetsskäl är denna i Uppsala-BBN-LISP-systemet utförd med hashningsmetodik. Man vet att vissa atom-teckensträngar är särskilt vanliga, samtidigt som symboltabellen blir ganska stor. Bara systemet tar upp (storleksordning) 1000 atomer, och därtill kommer allt som användaren inför. Det är då intressant att veta hur sned fördelningen mellan olika atomer är, vilka som är särskilt vanliga, och huruvida man skulle kunna fånga dessa i en "förhashning" före den normala genomgången av symboltabellen. Denna uppgift lades ut som ett trebetygsarbete. Rapporten från detta arbete refereras här i sin helhet.
Översikt. Hashmetoder används för att snabbt leta reda på lagrade data som identifieras av en teckensträng (nyckel). Nyckel och data, som kan vara en pekare till en större mängd data, lagras tillsammans i en hashtabell på en plats som bestäms ur nyckeln genom en lämpligt vald funktion $H_0=H_0(KEY)$ (hash-funktionen). Denna ger för varje nyckel en plats i tabellen men en plats kan motsvara ett stort antal nycklar. När man jämför den aktuella nyckeln med den som finns lagrad på platsen och finner att de är olika (kollision) måste man istället använda en funktion $H_n(KEY)$ efter den n:te kollisionen för samma nyckel.

För att lätt få en överblick över vilka platser i tabellen som genomlöpas av H_n betraktar man $S_n=H_n-H_{n-1}$ som ett steg och får $H_n=(H_{n-1}+S_n)$mod L där L är hashtabellens längd.

Man skall alltså välja två funtioner: H_0(KEY) och S(N,KEY). Detta val blir beroende på önskad tabellstorlek, nycklarnas antal, allmänna utseende och fördelning samt krav på stegantal, beräkningstider mm. I tabell 1 ges några kombinationer av L,H_0 och S. De sifferkombinationer som anges är nummer och årgång av tidskriften Communications of the ACM där de olika metoderna finns behandlade.

Några exempel: Div-rest metoden: Av nyckeln tillverkas ett tal D som för det mesta är mycket större än L. Sedan beräknas Q(KEY)= D/L och R(KEY)=Dmod L. Som H_0 används R, och Q används i vissa fall vid stegningen. D fås t ex genom kvadrering av korta strängar eller addition av delar av längre strängar.

Linjär sökning: S väljes så att det inte finns några gemensamma primtalsfaktorer hos S och L.

Kvadratisk sökning: $S=a+n.b+n^2.c$ är lättare än det ser ut. Kvadreringen åstadkommes genom addition av konstant till ett indexregister som för varje steg adderas till H_n. En nackdel är att endast halva hashtabellen kan genomsökas.

Slump-sökning: S fås ur en pseudoslumptalsgenerator med period L.

Viktad linjär sökning: S beroende av nyckeln men ej av stegnummer. Ex p=Q(KEY) enligt ovan eller p=H_0 i S=(2p+1)mod L i tabell där $L=2^r$.

Stegantal för några olika metoder: En teoretisk analys visar att antalet steg vid stora tabeller kan uttryckas som funktion av x=antalet upptagna platser/L. Sätt A(x)= antalet steg som behövs när en enskild plats sökes och E(x)=medelvärdet för alla tidigare A(x). Då erhålles för helt slumpmässiga nycklar vid:

slumpsökning: $A(x)=1/(1-x)$ och $E(x)=-(1/x)\ln(1-x)$

som också approximativt gäller för viktade metoder.

linjär sökning: A(x)=samma och $E(x)=(1-x/2)/(1-x)$

Tabell över E(x)

x	slump-sökning	kvadratisk-sökning	linjar-sökning	optimering vid inläggn.
0.1	1.05	1.05	1.06	1.05
0.3	1.18	1.19	1.21	1.16
0.5	1.39	1.44	1.50	1.28
0.7	1.72	1.84	2.17	1.45
0.9	2.56	2.79	5.50	1.80
0.99	4.65		(50.5)	2.24
1.0				2.5

Metoden "optimering vid inläggningen" finns beskriven i 2-73 av tidigare nämnda tidskrift, och bygger på viktad linjär sökning men med förflyttningar av redan inlagda nycklar om det skulle visa sig fördelaktigt. Inläggningen går långsamt men åtkomsten går desto snabbare.

Man bör observera att stegantalet ej är det allena saliggörande. Tiden för varje steg, inklusive tiden för jämförelse av nycklarna, måste också beaktas. Genom att välja tabellen mycket stor kan man nästan helt eliminera kollisioner och då blir ju stegtiden helt avgörande.
Man bör vidare observera att, när hashning användes på datamängder där nycklarnas frekvens varierar, de vanligaste har större chans att hamna på bra platser eftersom de i medeltal påträffas i början när tabellen är nästan tom. Detta medför att medelvärdet (viktat) för samtliga sökningar blir mindre än de rena medelvärde. En mindre del av detta arbete har varit att skriva en hashrutin som utnyttjar viktad linjär sökningsmetoden $S=2H_0+1$ till en tabell med $L=2^r.H_0$ beräknades genom addition och skrift av strängens bokstävers EBCDIC-representation. Vid testkörningar erhölls resultat enl. följande:

x	teor.E(x)	erhållet	viktat E(x)	högstantal steg
0.37	1.25	1.31	1.16	7
0.58	1.50	1.60	1.30	9
0.87	2.35	2.63	1.74	32

(L=1024)

Kända nycklar. Om man känner alla (eller större delen av) nycklarna i förväg kan man anpassa H_0 och S så att ett mindre antal kollisioner inträffar. En metod att göra detta är att konstruera en funktion H_0(KEY), parametrar),bygga motsvarande hashtabeller med de kända nycklarna för olika parameterkombinationer och välja den bästa.

Ex $H_0=\Sigma(2^{q_m}+1).C_m$ mod L där C_m är den m:te karakt. i strängen

q_m är m:te parametern

som realiseras med skift och additioner. Det gäller dock att inte göra H_0 så komplicerad att vinsten försvinner genom långa beräkningstider.

Vid inläggning i fix tabell bör man ta nycklarna så att de vanligaste kommer först vilket minskar åtkomsttiden.

Tillämpningen: LISP-systemet. För ett fall där hashtabellen är mycket stor och ligger på ett antal page-ar vill man förkorta åtkomsttiden genom att lägga de vanligaste nycklarna i en särskild för-hashtabell så att man slipper många tidskrävande page-förflyttningar. Avsikten är att göra ett försök i denna tabell och om det misslyckas gå vidare till den ordinarie tabellen.

Först gjordes statistik på nycklarna (ca 83 000 från olika LISP-dataset) och resultatet finns i tabell 2. De vanligaste nycklarna har en frekvens på några få procent och för mera ovanliga är frekvensen $f(n)=0.15/n$ där n är ordingsnummer i frekvensordning. Andelen enbokstavs-nycklar var ca 20% och andelen för de 48 vanligaste flerbokstavs-nycklarna var ca 35%. Eftersom de 48 nycklarna varierar mest i de sista bokstäverna valdes hashfunktionen $H_0=(2^{p_s}+k_s)C_s+(2^{p_n}+k_n)C_n+(2^{p_m}+k_m)M$ där p_s,p_n och p_m åstadkommer skift för sista,nästsista bokstaven och stränglängden M.k_i kan anta värdena 0 och 1. Den bästa påträffade parameteruppsättningen var: $p_s=5$,$p_n=p_m=3$,$k_s=k_n=k_m=1$ som tillåt 40 av 48 nycklar gå direkt in i en tabell med längden 64. Även för längderna 128 och 256 var denna uppsättning bäst. Vid försök med andra nycklar erhölls naturligtvis något sämre resultat. Vidare har Assembler-rutiner konstruerats som bygger en för-hashtabell av önskad längd: BUILDHASH(LENGTH;nycklar och data), adderar ytterligare nycklar:

ADDHASH(TAB-ADR,nycklar och data), utplånar tabell:DELETEHASH(TAB-ADR) samt anknyter tabell till ordinarie hashtabell: SETHASH(TAB-ADR).

Tabell 1. Olika kombinationer av H_0,L och S

Tabellstorlek	Godtycklig	Primtal	Tvåpotens 2^r
H_0	div-rest	div-rest 1-68	maskning (=div-rest)
$S=S_0$=konst.	RP	linjär sökn. S=godtyckligt 1-68	S= 2p+1 p=godtyckligt
S=S(M)		kvadratisksökn. $S=a+b.n+c.n^2$ 1-68,2-70,8-70	slumptalsgener. 1-68
S=S(KEY)	RP	viktad linjsökn med S=Q(KEY) 11-70	vikt.linj.sökn med S=2p+1 p=p(KEY) t ex $p=H_0$ 12-72
S=S(N,KEY)		viktad kvadr.sökn. $S=a+b.n+\frac{1}{2}Q.n^2$ 2-70	

anm. RP=tal utan primtalsfaktorer gemensamma med L

Tabell 2. De 48 vanligaste flerbokstavs-nycklarna.

nyckel	%	nyckel	%
QUOTE	5.0	PRIN1	0.4
SETQ	3.4	IN	
LAMBDA	2.6	TMP	
CAR	2.3	NPROP	
COND	2.2	APPLY	0.3
CDR	1.7	RPLACA	
NIL	1.2	STACK	
CONS	1.1	NLISTP	
RETURN		NOT	
LIST		PROGN	
AND		NLAMBDA	
NULL	0.9	RPAQQ	
GO		MEMB	
OR	0.8	TERPRI	
EQ		RPLACD	
PROG	0.7	OF	
FN		PUTDQ	
CADR	0.6	SELECTQ	
FUNCTION		IPLUS	
IS	0.5	FILE	
ATOM		GET	
PRINT			
TO	0.4		
LP			
THE			
CAAR			
CDDR			

4c. NovaLISP

Ett LISP-system, t ex det ovan beskrivna INTERLISP-systemet, innehåller en ganska stor mängd hjälpprogram och verktyg för programuttestning. All denna kod ligger i den virtuella adressrymd som användaren hela tiden har direkt tillgänglig. På en större maskin med flera LISP-användare är denna organisation lämplig, även om varje användare bara använder en mindre del av de tillgängliga redskapen, eftersom de gemensamma delarna av det virtuella datastrukturminnet bara lagras i en kopia, och delas mellan användarna. Den gemensamma minnesdelen kan då karakteriseras som ett programbibliotek, som hela tiden under körsessionen finns omedelbart tillgängligt till alla användare.

Samtidigt innebär denna organisation svårigheter om man vill köra LISP på en mindre maskin, t ex i ett enanvändarsystem. Detta visade sig vid DLU under tidigare år under arbetet på ett INTERLISP-system för Siemens 305 (jfr tidigare årsredogörelser). Samtidigt finns det problem som lämpar sig för programmering i LISP, och som är så begränsade att de borde kunna exekveras på en minidator, om man bara inte behövde släpa med alla hjälpprogram i systemet.

En möjlig metod är då att testa ut och kompilera programmet på ett fullständigt LISP-system, dvs på en större maskin, och därefter exekvera det på en minidator. Denna metod underlättas av att LISP-kompilatorn först genererar en sekvens av makroanrop, ett slags maskinkod för en tänkt LISP-maskin, och sedan bryter ned denna till egentlig maskinkod för resp. maskin. Man borde alltså kunna gå in i LISP-kompilatorn för 360-LISP-en och byta ut lågnivåsteget mot ett alternativt sådant som expanderar makrona till kod för en minidator. För att kunna exekvera denna kod krävs vidare att LISP:s centrala "runtime-system" (dvs rutiner för in- och utmätning, minnesallokering, garbage collection, odyl.) också skrivs för minidatorn, och kan anropas av den utlagda koden.

Under senare delen av 1973 utfördes dessa uppgifter för en minidator av typ Nova, som innehas av institutionen. Avsikten var att få en uppskattning av svårighetsgraden, tidsåtgången, och eventuellt oförutsedda svårigheter i sådant arbete, snarare än att få ett system för praktisk användning. Erfarenheterna skulle komma till användning vid planering av kommande projekt, där denna metodik kunde komma i fråga. Runtime-systemet har byggts ut till att innehålla ett tillfredsställande antal faciliteter, kompilatorn har modifierats, och ett antal små testexempel har körts. Däremot har vi inte definierat alla de makron, som kompilatorn kan definiera, utan endast de som behövdes för dessa testexempel. Övriga makron har endast studerats på papperet för att söka eventuella svårigheter.

Slutsatsen av arbetet är att ett specialiserat LISP-system av denna typ för en liten maskin kan skapas med en arbetsinsats på ca 4 manmånader.

Härefter följer en beskrivning av det framtagna systemet. Korskompilatorn har två steg:

Kompilering. Av LISP-koden generas LAP-kod (maskinoberoende assembler)

Korsassemblering. Av LAP-koden genereras den maskinkod som skall exekveras.

LISP-kompilatorn innehåller bl a själva kompileringssteget och en funktion LAP som anropas mellan de två stegen. LAP har omdefinierats så att den i stället anropar en korsassemblator till Novan. I assemblatorn finns definierade de makros som är nödvändiga för nedbrytningen från LAP-kod till Nova-assembler.

Ex. Kompileringssteget genererar av s-uttrycket (NULL x)

Lap-koden (LDV X)

(JNN 3).

(LDV X) står för: Ladda register 1 med pekare till variabeln x:s värde. X:s värde (bindning) har lagts i en argumentstack vid anropet av funktionen och finns på avståndet VREF från stacktoppen.

VREF definieras: (Antal argument till funktionen - X:s ordningsnummer bland dessa)

(LDV X) kommer således att vid anrop av makron LDV expandera till

(LDA 3 TOP + 8)
(LDA 1 -2 3)

där TOP+8 pekar på stacktoppen, register 3 sätts till basregister och -2 är X:s displacement relativt stacktoppen. Korsassemblatorn finns beskriven i avsnitt 3b.

Inskränkningar av reglerna vid kodning i LISP: Globala variabler får ej användas, antalet argument vid anrop av funktion måste vara korrekt. Korskompilatorn har tre fasta argument:

1. Loccount:s värde vid assembleringens början.
2. Argumentstackens djup vid assembleringens början.
3. Symboltabell.

Tillkommer en uppräkning av de funktioner som skall kompileras.

För varje funktion som kompileras printas LAP-koden, en symboltabell, felmeddelanden, maskinkoden och en lista över de s-uttryck som skall pushas i argumentstacken. I "top-loopen" till runtimesystemet finns en funktion som med Novans residenta Binary läser in maskinkoden från hålremsa, lägger den på avsedd plats i minnet och läser in och pushar eventuella s-uttryck.

Runtimesystem upptar 2k ord och innehåller förutom ovan nämnda "top-loop" funktionerna Read, Print och Cons med garbage collection. Till Read och Print finns Mkn (box) och Unbox som sköter omvandlingen mellan heltal i teckenform och heltal lagrade enligt lisp-systemets konventioner. Då Novan endast har 8k ords minne har de kompilerade funktionernas utrymme begränsats genom att så mycket som möjligt av koden lagts in i run-timesystemet. Vid initieringen av systemet sätts parameter till önskad längd av de olika delfälten: Icke numeriska atomer, parameter och kontrollstack, stora heltal och printsträngarea, kompilerad kod och liststrukturer samt inläsningsbuffert.

Under testkörning har 200 ord tilldelats varje delfält utom till fältet

för kompilerad kod och liststrukturer som har tilldelats 4400 ord exklusive det utrymme som den kompilerade koden tar. Varje lispcell representeras av två helord vilket ger 2200 lispceller. För evaluering av funktioner läsas dessa in kompilerade i systemet varefter kontrollen överlämnas till den första av dem. När de exekverat klart återfår "toploopen" kontrollen och nya funktioner kan läsas in för exekvering.

Status: Systemet är testat och enkla kompilerade funktioner har med lyckade resultat körts i systemet.

ex. (DE INARG NIL (PRINT(NAPPEND(READ)(READ))))
(DE NAPPEND(X Y)(COND(NULL X)Y)(T(CONS(CAR X)(NAPPEND(CDR X)Y)))))
där INARG har fått kontrollen av "toploopen".

4d. Effektivitetsmätningar i LISP F1.

Inledning

Under vårterminen företogs en effektivitetsstudie av LISP F1 (Utförligare om LISP F1 finns i nästa avsnitt). Målsättningen var att inför en kommande omskrivning av LISP F1 få klarhet i, vilka rutiner som krävde speciell uppmärksamhet av effektivitetssyfte. Dessutom var det intressant att få klarhet i misstanken att:

- FORTRAN:s I/O stjäl mycket tid;
- sökning i atom-tabellen ineffektivt.

Metodbeskrivning

En interupt rutin skrevs i assembler, som fick kontrollen vid bestämda tidsintervaller. Tidsintervallet var 0.027 sek. (Statistiskt korrektare hade det varit att slumpa tidsintervallet, men det visste jag inte då). Vid varje interupt anropades en FORTRAN-rutin. Som argument gavs den absolutadress, där avbrottet skett. En tabell över adressintervall i LISP F1 uppdaterades, och efter avslutad körning skrevs tabellen ut på disk. Statistikprogrammet fanns med i LISP F1 tills 500 körningar gjorts, och den statistik som presenteras nedan täcker ett brett spektrum av LISP-körningar, från elev-körningar till rena produktionskörningar.

Några ord om hur adresstabellen skapades kan vara på sin plats. LISP F1 består av ett huvudprogram på ca 1000 statement om fattande eval-apply samt alla subr'ar och fsubr'ar. Detta kan ej enkelt styckas upp på grund av den rekursion kodningstekniken. Dessutom finns ett antal subrutiner, framförallt för I/O. Subrutinnamnen deklarerades som EXTERNAL i en assembly-snutt, som lade dessa adresser i en tabell och skickade tabellen till en FORTRAN-rutin. Huvudprogrammet delades upp i lämpliga avsnitt, och varje avsnitt började med ett stmnr. I en initieringsfas av huvudprogrammet lades kodsekvenser:

```
ASSIGN stmnr TO N
ADRVEC (1)=N
ASSIGN stmnr TO N
ADRVEC (2)=N          osv.
```

ADRVEC kompleterades sedan med adresserna från assembly rutinen och sorterades slutligen. Ett försök att komma åt adresserna inne i huvudprogrammet genom att stoppa in en massa ENTRY-satser misslyckades, då kompilatorn lade ut alla ENTRY-adresser sist i programmet, följt av initieringskod (med bl a hopp till den "verkliga" ENTRY-adressen).

Resultattabeller

(Kolumnen "Ny version" förklaras först i avsnitt "slutsats").

Antal mätningar = 500 st.

Tabell 1. Översikt

rutin	tid i %	standard avvikelse	Ny version
EVAL-APPLY	9.239	5.449	
MATCH[1]	1.094	0.722	
SUBR,FSUBR[2]	1.525	1.080	
PUSH,POP	19.850	11.830	
INPUT-rutinen	27.535	18.017	
OUTPUT-rutinen	7.395	6.475	
GBC[3]	3.207	3.983	
FTN I/O	12.681	17.222	0.144
OK[4]	4.082	3.644	1.306
CONS	4.868	3.799	3.373
MAKFREE[5]	3.101	5.574	
INIT[6]	1.290	3.635	
	S:a 92.660		
	Övriga < 1.000		

[1] MATCH	de kodavsnitt, som avgör till vilken SUBR,FSUBR man skall hoppa.
[2]SUBR,FSUBR	alla FORTRAN-kodade LISP-funktioner utom COND och CONS.
[3]GBC	Huvudrutiner för garbage-collection. Anropar MAKFREE.
[4]OK	En rutin som bara kollar att argumentet ligger i ett visst tal område (är en cons-cell)
[5]MAKFREE	Gör fri lista. Anropas från GARB och INIT:
[6]INIT	Initierar LISP Fl. Anropar MAKFREE.

Tabell 2. INPUT

rutin	tid i %	standard avvikelse	Ny version
READ[1]	0.613	0.714	
MATOM[2]	15.455	10.388	1.106
RATOM[3]	1.013	1.034	
JATOM[4]	4.561	3.862	
SBYT[5]	1.827	1.799	0.961
SHIFT[6]	4.065	3.477	1.360

[1]READ	huvudrutinen för LISP:s read funktion
[2]MATOM	söker inläst atom i atomtabellen
[3,4]RATOM" JATOM	läser en atom
[5]SBYT	lagrar ett tecken i buffert till MATOM
[6]SHIFT	läser nästa tecken från input-strängen.

Table 3. OUTPUT

rutin	tid i %	Standard-avvikelse	Ny version
PRINT[1]	1.037	1.173	
PRINAT[2]	2.371	2.087	
PCHAR[3]	3.351	3.206	1.836
OUTSTR[4]	0.636	0.645	

1. PRINT huvudrutin för LISP:s print
2. PRINAT lägger en atom i outputbufferten. (sköter även om "prettyprint")
3. PCHAR lägger ett tecken i outputbufferten
4. OUTSTR skriven ut outputbufferten

Tabell 4. GETCH, PUTCH[1]

rutin	tid i %	Standard avvikelse
GETCH	0.686	0.924
PUTCH	0.861	0.892

1. GETCH, PUTCH assemblykodade rutinen för flyttning av ett tecken från ett ord till ett annat.

Kommentarer till tabellerna

Misstanken om att FORTRAN:s I/O är tidskrävande var riktig (12.681 % tabell 1). Att den linjära sökningen var så ineffektiv var nog en liten överaskning (15.455%, tabell 2). F ö var det oerhört intressant att få bekräftat, att de triviala småsnuttarna stal så mycket tid. Se t ex SHIFT (4.065%, tabell 2) och PCHAR (3.351%, tabell 3). Lägg också märke till den ringa tid som åtgått i de assembly-kodade GETCH och PUTCH (0.686%, resp 0.861%, tabell 4). Rutinen OK (4.082%, tabell 1) är också ett typexempel på var tidsbesparingar kan göras (OK signalerar fel, om argumentet ej är en cons-cell. Detta har inträffat ytterst sällan om testen ligger numera inline vid ett fåtal känsliga ställen). Det kan också noteras, att max tiden för alla FORTRAN-kodade LISP-funktioner (förutom CONS, som används flitigt även av systemet) var 0.451% (COND), därefter följer PROG (0.291), SETQ (0.203), övriga <0.1.

Slutsats

Skriv egen I/O ist.f. FORTRANS I/O. (All grundläggande I/O i LISP F1 är trivial). Skriv smårutinerna för teckenhantering och triviala tester samt CONS i assembler. Organisera atom-tabeller som en hash-tabell. Detta är numera gjort i LISP F2 och kommenteras närmare i nästa avsnitt. Resultatet kan också ses i tabellerna under kolumnen N·y version.

4e. LISP F1 och F2

(a) Bakgrund. LISP F1* (och LISP F2) är en kärnminnes resident LISP-interpretater, skriven i FORTRAN. Programmet har funnits i drift vid DLU sedan slutet av 1970 och har för (1974-01-15) distribuerats till 76 användare, varav 52 st. till USA. Interpretatorn är ett (litet) a-liste system (variabellindningar på en associationslista) med några moderna, BBN-inspirerade faciliteter som:

---

* LISP F1-systemet beskrivs i en manual: Diz Breslaw, Mats Nordström, Erik Sandewall, LISP F1 Users' Manual, Datalogilaboratoriet 1970.

implicit progn i λ-uttr och cond no-spread och half-spread λ-uttr. break-funktion (spec. för interaktivt bruk). Vad som saknas är framförallt: flytande tal; stora heltal; arrayer; möjlighet att kompilera.

(b) Utvecklingsarbeten under 1973

Under vårterminen fullföljdes följande 3-betygsarbeten:

- Byt av FORTRAN I/O mot assembly-kodade I/O-rutiner. ref 2
- Atomtabellen som hash-tabell samt atom-GBC. ref 2
- GBC-skyddad minnesaren (för t ex funktionsdefinitioner) ref 3
- Kompakterande GBC samt minnesdump av det kompakterade utrymmet ref 3

Dessutom har en omfattande effektivitetsanalys av LISP F1 företagits (se kap. 4d, denna rapport).

I augusti var det dags att slå ihop alla nyheter + erfarenheter från tidsstudier + en del andra nyheter, och eftersom interpretatorn är måttligt stor (ca 3000 FORTRAN-satser) skrevs hela systemet om från grunden och döptes till LISP F2. Speciell hänsyn har tagits till att få systemet så lätt-implementerat som möjligt på andra maskiner, oberoende av ordlängd, I/O-rutiner m m.

(c) Nyheter i LISP F2-systemet

(i) Godis till användaren:

5-10 ggr snabbare än LISP F1
~1.5 ggr långsammare än Stanford-LISP, skriven i assembler)
Snyggare prettyprint
Rollout/in oberoende av kärnminnesregion
GBC-skyddat minnesområde
Atom-GBC
Relativt stort LISP-minne under TSO (14000 LISP-celler)
Nya felutskrifter samt fullständig kontroll av felhanteringen
Fullständig kontroll av teckeninläsningen (Egna brytkaraktärer
En del nya vanliga LISP-funktioner, som t ex mapcar, selectq m m.

(ii) Förenklingar för implementatören

En rutin (INIT) består enbart av de variabler (13 st) som

behövs sättas beroende på maskinstorlek och typ. F ö måste man ändra i common-blocket (som är gemensamt för alla rutiner) samt skriva en trivial assembly-snutt för flyttning av ett tecken någonstans. Detta är allt som behövs för att starta upp systemet. Dessutom gäller att:

- All användning av FORTRAN:s I/O sker via 11 små rutiner, som bara innehåller var sin I/O sats. Detta för att förenkla utbyttet av dessa mot egna assemblykodade varianter.
- Tips ges, vilka korta rutiner, som bör skrivas i assembler för att snabba upp systemet (rutiner i stil med PUSH, POP).
- När systemet är i drift, kan hela initieringsrutinen (INIT 2) ersättas med CALL ROLLIN (n), varefter de standard funktioner m m man tidigare dumpat, är tillgängliga redan vid starten.
- Ändringen batch-mode/interactive mode sker genom att under körning evaluera (SETBIT 8) resp (CLEARBIT 8)

(iii) Effektiviseringar i koden

- Vid anrop av subr'ar göres ej eblis, utan argumenten evalueras och läggs på en stack.
- Kritiska push och pop ställen har kodats in-line.
- Smårutinen (push-pop samt grundläggande I/O) har skrivits i assembler.

(d) Nuvarande status och framtida utvecklingar

I Uppsala finns LISP F2 tillgängligt i två versioner. En 104K-bytes version (för TSO-körningar) och en 208K bytes (för batch körningar). Inga fel är kända för nuvarande. En preliminär ändringsdokumentation finns i form av en stencil. En ny manual planeras (i mån av tid och externt intresse). F ö finns för tillfället inga planer på ytterligare arbeten på LISP F2.

4f. PLAST

Programmeringsspråket PLAST har kortfattat presenterats i DLU:s förra årsrapport. En kort repetition av syftet med språket kan dock vara på sin plats:

I PLAST vill man förena LISP:s karaktär och kraftfullhet vad det gäller list- och symbolbearbetning med en enkel, BASIC-liknande notation för detta ändamål.

Arbetet med PLAST och åtgärderna kring detta har under det gångna året i huvudsak bedrivits inom tre områden av olika omfattning:

I. Intresse - behov - krav.

I ref 1, "Utvärdering av enkätundersökningar angående små databaser och PLAST" har ett försök gjorts att utifrån ett litet material av enkätsvar sammanställa de synpunkter på smådatabasteknik och PLAST, som eventuella avnämare i industrin och forskningsmiljöer lämnat. Två olika undersökningar syftade till att dels bedöma ett antal föreslagna tillämpningar och ge förslag på ytterligare tillämpningar inom forskningsområdet smådatabaser, dels bedöma PLAST ur ett antal olika aspekter. Av de ca 100 tillfrågade personerna svarade tyvärr endast omkring 20, men trots det bräckliga utvärderingsunderlaget bör några resultat av undersökningen kunna vara av värde:

- Ett språk av PLAST-typ borde vara användbart i minst lika hög grad som andra språk för att lösa smådatabasproblem.
- PLAST förefaller vara lättlärt och lätthanterligt och tillräckligt kraftfullt för sitt ändamål.
- Det anses vara av stor betydelse att språket är interaktivt tillgängligt och att det ges möjlighet till kommunikation med en stor databas.

Det förtjänar att påpekas, att praktisk erfarenhet av programmering i språket inte ligger till grund för värderingarna. Andra synpunkter, såväl positiva som negativa förekommer, men det allmänna och slutgiltiga intryck som undersökningen ger, är att det uppmuntrar till fortsatt experiment- och utvecklingsarbete.

II. Tillämpningsexempel.

Ett par månander ägnades åt att programmera upp ett exempel på användning av PLAST. Detta program presenteras i avsnitt 12b av denna rapport.

III. Implementeringsarbete.

Förverkligande av tidigare planer.

Av de vidare utvecklingsplaner som framlades i förra årets DLU: rapport

har den, som avser effektivare PLAST-implementeringar varit den mest angelägna och har tagit den mesta tiden i anspråk (se kommande avsnitt). Syntaxfrågor har diskuterats och diskuteras nu och då, men några radikala idéer till alternativ förefaller inte vara aktuella. Möjligheten att effektivisera den genererade interna koden har undersökts, och resultatet pekar på, att det med ganska liten arbetsinsats kan tillföras systemet faciliteter för en enkel form av optimering av koden. Praktiska prov har visat, att en inte föraktlig tidsvinst kan uppnås med dessa små medel. I och med att en ny parser tagits i anspråk för inöversättning av PLAST-uttryck (se mera om denna senare), förefaller också problemet att låta användaren definiera egna syntaxutvidgningar bli avsevärt mycket lättare att lösa.

Erfarenheter av tidigare implementation.

Implementationen av den ursprungliga, experimentella versionen av PLAST (fortsättningsvis kallad X), vilken beskrivits i tidigare årsrapporter och som dokumenterats i ref. 2, får nu anses vara avslutad. Systemet, som alltså är byggt på FORTRAN-LISP-interpretatorn, innehåller endast de centrala delar av språket som krävs för att testa dess notation och struktur. Sådana yttre faciliteter som t ex filhantering och editering har inte ansetts nödvändiga för att uppfylla dessa minimala krav. Dessutom är X-versionen batchorienterad, vilket ytterligare minskar betydelsen av sådana påbyggnader på detta stadium, i synnerhet som avsikten med PLAST ju är att vara ett interaktivt språk. De användare (främst 3-betygsstudenter), som hittills begagnat PLAST har inte framfört någon omvälvande kritik mot det som systemet avser testa, och med endast smärre modifikationer bör de idéer som präglat X-implementationen kunna ligga till grund för fortsatt utvecklingsarbete.

De slutsatser som kan dras av den grundläggande experimentverksamheten är alltså först och främst, att den föreslagna strukturen och notationen för PLAST väsentligen bör bibehållas.

Nya implementationer.

I samband med att BBN-UPPSALA-LISP (INTERLISP) började kunna användas och DLU fick tillgång till QZ:as DEC-10-maskin Simon via terminal, påbörjades nya PLAST-implementationer, en i vardera INTERLISP och Simons Stanfordlisp. Med erfarenheterna från X-systemet gick kodningen av de

centrala delarna ganska snabbt. En del nya principer och tekniker utnyttjades dock. Under det att X-versionen till väsentliga delar (framför allt översättaren) hade kodats i FORTRAN, vilket var relativt tidskrävande, skrevs nu hela systemet i LISP, och därmed kunde uttestningstakten ökas högst väsentligt. Den nya tekniken utgörs av att den parser (IKP, se ref 2), som ursprungligen använts för internalisering av PLAST-uttryck i Simon-PLAST bytts ut mot TOP-parsern (se avsnitt 4g och 9c). Erfarenheterna av denna blev så god, att även INTERLISP-PLAST avses skrivas om med användande av TOP. Fördelen med TOP framför IKP på det parsetekniska planet ligger främst i att nyckelordssekvenser (t ex IF ... THEN ... ELSE-satser) kan hanteras mycket enklare. Dessutom tar TOP mindre plats och är snabbare än IKP, vilket ytterligare stärker skälen för den. Sedan kärnorna i det nya systemet testats ut, avtog arbetet med INTERLISP-implementationen inför konkurrensen med den mera inspirerande och effektivare interaktiva miljön kring Simon-PLAST. Här utökades ramen runt språket med några ytterligare möjligheter, t ex med elementära kommandon för att lagra procedur-definitioner på och ladda in dem från en yttre fil. Slutligen kompilerades hela koden till en laddmodul, som utgör ett väl fungerande och rimligt effektivt PLAST--system.

Planer.

Den angelägnaste uppgiften i den allra närmaste framtiden är att förse framför allt Simon-PLAST med möjligheter till editering av procedurer. Diskussioner om hur ett sådant editeringsspråk bör konstrueras pågår i skrivande stund. Med den utbyggnaden är det vår förhoppning att kunna presentera PLAST-systemet även för andra Simon användare och intressera dessa för att pröva språket enligt deras egna förutsättningar och behov. I samband med denna utvidgade test- och utvärderingsverksamhet är det också angeläget att ganska snart kunna tillhandahålla en reviderad PLAST--manual, som vänder sig till användare mindre förtrogna med listprogrammeri INTERLISP-PLAST skall så vidareutvecklas till att nå samma status som Simon-PLAST, så att båda systemen blir helt kompatibla och så att utvecklingen kan drivas parallellt på två fronter.

Vidare är en väsentlig arbetsuppgift att implementera ett kraftfullt

filhanteringssystem för såväl program som data, och för detta ändamål verkar INTERLISP:ens metoder attraktiva.

På något längre sikt förefaller det också naturligt, att bereda möjligheter för PLAST-användare att kunna kompilera sina program för att uppnå högre effektivitet. I detta sammanhang bör naturligtvis de experiment med optimering av koden, som tidigare nämnts, åter tas upp.

Förhoppningsvis bör det inom ramen för vårterminens arbete även finnas utrymme att förverkliga idéerna om en syntaxutvidgningsfacilitet.

Referenser

1. Olle Willén, Utvärdering av enkätsundersökning angående smådatabasen och PLAST, DLU, juni 1973. (DLU 73/13)
2. Olle Willén, PLAST: dokumentation av den experimentella implementationen, DLU, januari 1973. (DLU 73/2)

4g. REC.

REC är en implementation av några, helt eller delvis, nya idéer för programmeringsspråk, speciellt den variabelfria kalkylen.

Karaktäristiska drag hos REC.

1. En-en-tydig motsvarighet: flödesschema - program - intern struktur.
2. Operatorsyntax.
3. Godtycklig vänstersida i tilldelningssatser.
4. Go-to-fritt språk.
5. Bygger på den variabelfria kalkylen.

Flödesschema - program - intern struktur.

REC är väl lämpat för programmanipulation, då det har program som datatyp. Ett REC-program ligger internt lagrat som programmets flödesschema. Genom att det finns en en-en-tydig motsvarighet mellan flödesschema - program - intern struktur, så är det lätt att manipulera på program, att bygga upp nya program, att exekvera dessa och att skriva ut dessa.

Ett exempel: uttrycket f(a, b, c) får man av:

ser('f, par('a, par('b, 'c))).

De program-(funktions-)datatyper som finns är: ser - f(g), par - (f, g), cond - (if f then g else h), prog - (f; g), infix - (f + g), quote - ('f), subr - (maskinkod), label - (f: används när man vill sätta ett namn på en struktur).

Dessutom finns datatyperna: int - (heltal), real - (reella tal, ännu ej implementerade), char - (karaktär), string - (sträng), array - (vektor), tab - (binära söktråd).

Till varje datatyp finns sedan en konstruktionsfunktion med samma namn och som ger ett objekt av motsvarande typ!

Operatorsyntax.

Syntaxen för ett REC-program är följande!

<obj>::=<elemobj>|<rightop> <obj> | <obj> <leftop> |
<obj> <binop> <obj> | <obj> <obj>
<rightop> ::= (| if | ' | + | -
<leftop> ::=)
<binop> ::= ; | , | : | then | else | := | and | or |
= | > | < | + | - | * | / | **| . | " | "
<elemobj> ::= grundobjekt, såsom tal, karaktärer, identifierare, strängar m m.

Som synes är det en ren operatorsyntax. Trots detta ser ett REC-program nästan exakt ut som ett algol-program.

TOP.

Till denna syntax har en mycket enkel, generell parser utvecklats, kallad TOP (Torgnys Operator Parser). I sin senaste version ser den ut som (uttryckt i REC):

'parse: right(x, rdobj) |
'right: if empty getright then getnext; left else
left(x, func . y) |
'left: if x > prior then y else
left(x, func . y) |

```
´getright: rest(func := get(rightab, obj)) |
´getnext: (if empty then prior := 17;
                func := ´ser(x, right(16, copy obj)) else
                prior := x0; func := x1)get(leftab, rdobj) |
```

Här är rdobj en grundfunktion som läser in ett objekt och ger obj detta värde. I leftab ligger alla operatorer som har vänsterprioritet, dvs binära och vänster-operatorer. I rightab ligger alla operatorer som saknar vänsterprioritet, dvs höger- och anadiska operatorer. (En anadisk operator är en nollargumentsoperator).

TOP fungerar nu så att så snart en operator påträffas i inputströmmen, så får operatorns konstruktionsfunktion kontrollen. Denna läser sedan in så mycket den vill ha, bygger upp det uttryck som önskas, och lämnar sedan tillbaks kontrollen till TOP.

För att läsa in lagom mycket använder operatorns konstruktionsfuntion parse-funtionen. Parse läser in tills den träffar på en operator med vänsterprioritet lägre än argumentet till parse. Alltså om parse anropas med ett lågt tal fås ett stort uttryck medan om den anropas med ett högt tal så fås ett litet uttryck.

Som exempel på en konstruktionsfunktion kan if:s funktion tagas.
I rightab under nyckel$_n$ if ligger:

```
cond(parse 2, parse 2, if obj = ´else then parse 2 else nil).
```

I leftab ligger under både then och else:

```
(1, nil).
```

Detta betyder att båda if - then och if - then - else uttryck kan läsas in. Men funtionen testar inte om det verkligen är then som följer efter if. Om man vill att felutskrift ska ges om det inte är then, så får man ändra andra argumentet till cond ovan till:

```
(if obj = ´then then parse 2 else error).
```

Syntaxutvigning.
Genom TOP:s enkla struktur är det mycket lätt att utvidga REC:s

syntax. Vill man t ex införa en for-loop med utseendet:

```
for i := 1 to 10 do foo |
```

så lägger man upp i rightab under nyckeln for:

```
ser('fora, lis(parse 2, parse 2, quote parse 2, 'id))
```

och i leftab lägger man (1, nil) under to och do. Sist definierar man fora såsom:

```
'fora: if x> y then nil else
            eval rest rest; x := x + 1; fora |
```

Om nu TOP läser in ovanstående for-uttryck så kommer detta att översättas till:

```
fora(i := 1, 10, 'foo, id)
```

När detta evalueras får man det man vill ha.

Så enkelt var det att utvidga syntaxen. För att utvidgningen ska bli fullständig så får man dessutom utvidga printfunktionen så att uttrycket fora(...) även blir utskrivet på formen for ... to ... do ...

Tilldelningssatsen.

I REC kan man ha ett godtyckligt uttryck på vänstersidan i tilldelningssatsen. Detta hänger ihop med att värdet av ett REC-uttryck alltid är en pekare till den cell som innehåller värdet av uttrycket.

Detta medför att om man gör ett uttryck A lika med ett uttryck B, dvs A := B, så är efteråt oftast uttrycket A lika med B, dvs A = B ger värdet true. Ett fall då detta inte gäller är (a + b) := 3 ty värdet av a + b förändras inte av detta.

Go-to-fritt språk.

De flesta programmeringsspråk har filosofin att ha en mycket restriktiv syntax, så att en massa programmeringsfel upptäcks vid kompileringstillfället. Nackdelen med denna filosofi är att många i och för sig legitima program blir otillåtna, så att man ofta måste krångla och trixa med en massa smådetaljer för att få det gjort som man vill ha gjort. Dessutom kan denna metod inte upptäcka de verkligt svåra felen, tankegrodorna - de logiska felen.

REC har en annan filosofi. Där är alla tänkbara programkonstruktioner tillåtna. I stället har språket en mycket enkel grundstruktur som gör att båda lätta och svåra fel blir lätta att bemästra.

Jämför man REC med algol så finns det i algol många olika men likartade begrepp. I REC finns det däremot bara ett fåtal, men kraftfulla, begrepp. I REC finns t ex inget goto, utan där representeras samma sak av ett funktionsanrop utan parametrar och utan återhopp. Likaså finns det inga procedurdefinitioner i REC, ty varje label i programmet definierar en procedur.

En av fördelarna med att ha proceduranrop utan återhopp i stället för goto är att man då måste sätta ut en högerparentes för att det inte ska bli något återhopp. Detta medför att en vänsterparentes måste sättas ut, och detta gör att man får sina program klarare strukturerade.

Variabelfria kalkylen.

Till skillnad från alla andra programmeringsspråk så bygger REC på den variabelfria kalkylen i stället för λ-kalkylen. Den variabelfria kalkylen utmärks främst av att inga dummy-variabler är nödvändiga vid funktionsdefinitioner.

Ett exempel: Antag att man har två två-ställiga funktioner f och g och vill bilda funktionen som är summan av dessa. I λ-kalkyl måste man då skriva:

$$\lambda(x, y)(f(x, y) + g(x, y)).$$

I den variabelfria kalkylen skriver man:

$$f + g.$$

Alltså betydligt enklare och klarare. Men om man så vill så kan man även i den variabelfria kalkylen skriva:

$$f(x, y) + g(x, y).$$

En skillnad till algol är att man hela tiden har tillgång till en omgivning när man evaluerar ett uttryck. Man har funktionerna x, y och z som plockar fram första, andra resp tredje elementet ur omgivningen. Dessutom finns id som returnerar omgivningen oförändrad,

och rest som returnerar den omgivning man får när man tar bort det första elementet. Jämfört med algol så ser en procedurdefinition ut som om man hade de formella parametrarna x, y och z fördefinierade i varje procedur.

Ytterligare en liten fördel med den variabelfria kalkylen är att funktionsanrop i ett program i vissa fall kostar mindre, genom att inga variabelbindningar behöver göras vid inträdet i funktionen.

Kompilatorn.

Under hösten 73 påbörjades arbetet med att göra en kompilator till REC-systemet. Denna ska kunna kompilera hela program, enskilda procedurer, enskilda satser eller delar av en sats.

Det som är gjorts hittills är ett RAP(Rec Assembly Program), som är en generell assemblator som lägger ut maskinkod. Dessutom har en kompilator gjorts, som lägger ut kod som gör precis samma sak som vid interpreteringen. Det som återstår är att få kompilatorn att optimera koden.

DEC-REC.

Under dec 73 har en LISP-implementering av REC lagts upp på DEC 1070 (Simon) i Stockholm. Detta är i stort sett ett fullständigt REC-system, men ganska ineffektivt, eftersom det sker en del dubbelinterpretering.

Fördelen med DEC-REC-en är att den är lätt att utnyttja interaktivt. Detta är till stor hjälp vid programmeringsarbetet.

Utrymmeskrav.

I REC-systemet på IBM-370 så har systemet följande storlek:

Maskinkod	8 K bytes
Stackar	8 K bytes
IO-buffert	3 K bytes
REC-kod	28 K bytes
Fritt minne resten	

Som synes behövs 47 K bytes för REC-systemet. Då är att märka att nästan alla I/O-funktioner, med symboltabell och allting, ligger i form av REC-kod, för att användaren ska ha möjlighet att modifiera dem. Så ett REC-system passar mycket bra för minidatorer, ty utrymmeskravet kan minskas betydligt.

5. Manipulation av FORTRAN-program

5a. Slutrapport om FFC

FFC (som utförligt beskrivits i förra årsrapporten, DLU-73) är ett program för konvertering av Fortran-program skrivna i CD3600-dialekten av Fortran till IBM:s Fortrandialekt (Fortran IV) för System/360. Programmet utvecklades vid DLU under 1971 och användes vid Uppsala Datacentral av såväl kunder som anställda i samband med maskinbytet i årsskiftet 1971-72.

I januari 1973 kunde all konvertering av Fortranprogram anses vara avslutad (och dessutom hade den för FFC ansvarige återkommit från militärtjänst) varför en undersökning av erfarenheterna med FFC företogs. Ett enkätformulär med frågor rörande användningen av FFC utspriddes bland kunderna och personalen på UDAC och personlig kontakt togs med några utvalda kunder. Trots att nästan ett år gått sedan merparten av konverteringarna genomfördes inkom ett tillfredställande antal enkätsvar.

Av det totala arbetet med FFC, inklusive undersökningen om användningen, kan följande slutsatser dragas:

1. Det var troligen ekonomiskt försvarbart att utveckla ett översättningsprogram helt bortsett från värdet av minskad irritation hos kunder, snabbare konverteringsarbete mm. Totalt anses FFC ha inbesparat minst ett manårs arbete, men detta är mycket svårt att uppskatta. Det är t.ex. nästan ogörligt att uppskatta hur många nya fel som skulle introducerats i program av mänskliga konverterare.

2. Det var fel att låta FFC förutsätta att indata bestod av felfria 3600-Fortranrutiner. Många körningar misslyckades pga syntaxfel i indata (halvkonverterade program, ofullständiga rutiner etc.)

3. Personer med god programmeringsdisciplin och tidigare kännedom om standard-Fortran och problemen med skillnader i Fortran-

dialekter hade mycket små problem med konverteringsarbetet och behövde oftast inte alls använda FFC.

För utförligare redovisning av enkätsvaren och erfarenheterna från arbetet med FFC hänvisas till slutrapporten (DLU 73/A4).

5b FAP

Som trebetygsarbete omkodades FFC att acceptera IBM Fortran IV som indata och ur denna som utdata producera IBM Fortran IV, dvs. att bryta ned koden och sedan bygga upp den igen. Programmet kallades FAP (Fortran Analysis Program) och skrevs för att användas som basprogram för ett antal Fortranbearbetande program. Tyvärr kvarstod i FAP nackdelen med FFC, nämligen att det förutsatte att indata var fritt från syntaxfel.

Det vissade sig inte erbjuda mycket större svårigheter att parsa IBM Fortran IV än att parsa CD-3600 Fortran.

5c Reflist

Det i förra årsrapporten beskrivna korsreferenslistegenererande programmet (ett trebytygsarbete med FFC som grund) anpassades under våren-73 till FAP och vidareutvecklades för att ge för IBM Fortran IV mer välanpassade referenslistor.

Programmet ger korsreferenslistor på variabler, rutinnamn, COMMON-blocknamn och satsnummer i Fortranrutiner. Korsrefernslistorna innehåller för varje namn uppgifter om typ, dimension, commonblocks-tillhörighet, equivalencedeklaration mm och ger sedan en referenslista uppdelad på assigned, referenced, actual param, input och output (med actual param menas att variablen står som ensam aktuell parameter i ett anrop och alltså kan få ett värde tilldelat i den anropade rutinen). Det går alltså att ur referenslistan få information om variabler som refereras utan att ha fått något värde tilldelat, något

som IBMs kompilatorer inte påpekar. (Vanligen beror sådana situationer på felstavning av variabelnamn.) Erfarenheterna från arbetet med detta korsreferenslisteprogram visar att en fullständig referenslista för större Fortranrutiner innehåller för mycket information för att vara överskådlig, samtidigt som man inte vill avvara någon information om de objekt man blir speciellt intresserad av. I batchmiljö är dessa krav oförenliga (om man inte har tid att vänta på omkörningar) men vid interaktiv bearbetning vore ett program som tog fram fullständiga korsreferenstabeller och sedan på användarens begären visade urval ur dessa mycket praktiskt.

Ett sådant interaktivt program skulle t.ex. kunna visa namnen på alla variabler som bara refererats en eller två gånger, alla som lästs in, alla som är parametrar till en viss rutin, alla som används som index etc. Det skulle kunna lista alla variabler deklarerade (eller implicit deklarerade) som heltalsvariabler, flyttalsvariabler, arrayer etc. Troligen skulle ett sådant referenslistprogram vara till god hjälp vid kontroll av och felsökning i program.

En annan viktig användning av korsreferenslistor är som programdokumentation. Vid underhåll och felsökning i "andras" program är det en mycket stor hjälp att känna till samtliga referenser till ett namn, så att inte ändringar får oönskade bieffekter. I kombination med väl valda variabelnamn är en sådan lista också till god hjälp för förståelsen av ett programs interna funktion.

5d Tidsanalysprogram

Som trebetygsarbete har FAP också modifierats för att lägga in räknarsatser i Fortranprogram för att ge tidsanalys av programmen med den metod som beskrivits av Knuth (Empirical Study of Fortran Programs) och som sedan vidareutvecklats av Dan Ingalls.

Programmet lägger vid denna typ av tidsanalys i varje rakt kodavsnitt in en sats som räknar upp en räknarvariabel med ett steg varje gång

kodavsnittet genomlöps. Varje kodavsnitt har sin egen räknarvariabel. Med ett rakt kodavsnitt menas ett kodavsnitt som det varken går att hoppa ut ur eller hoppa in i och som alltså antingen genomlöps helt eller också inte alls.

Då programmet exekverat färdigt kan man alltså genom att studera räknarvariablerna få reda på exakt hur många gånger varje sats exekverats. Ett analysprogram kan lätt generera en programlistning med antalet exekveringar angivna bredvid respektive sats. Om analysprogrammet dessutom känner till den ungefärliga tidsåtgången för respektive sats kan den totala tiden tillbringad i varje sats också skrivas ut.

Tidsåtgången för satser i Fortranprogram kan för de flesta satser uppskattas som en summa av en satsberoende grundtid och ett tidstillägg beräknat ur antalet i satsen förekommande variabler, antalet och typen av ingående operatorer och i satsen ingående anrop till standardfunktioner (av typ SIN etc). Naturligtvis kan detta inte ge en exakt tidsangivelse, bl.a. beroende på användningen av register för indexberäkningar, optimeringar av koden etc., men tidsangivelserna är tillräckligt noggranna för att man skall kunna få en uppfattning om var programmet tillbringar mesta tiden (och följdaktligen var det lönar sig bäst att försöka optimera programmet).

Programmet är färdigt men har inte ännu använts till någon systematisk undersökning av program. En intressant användning är att köra programmet parallellt med ett tidsanalysprogram av den andra typen (den som ger tidsavbrott med vissa mellanrum och ser var programmet då exekverade) för att få en jämförelse av resultaten och dessutom en inblick i hur stora kostnaderna för de båda metoderna är i förhållande till de resultat de ger.

5e. Standardfortran

Under året har bedrivits förstudier till ett program avsett att läsa godtyckliga Fortranprogram och i dessa finna alla konstuktioner som

strider mot standard-Fortran. Det är förhållandevis enkelt att finna rena "syntaxfel" i Fortranprogram, medan det är betydligt svårare att kontrollera att t ex användningen av variabler av olika typer sker enligt standarddefinitionens regler.

Arbetet blir speciellt intressant av det faktum att programmet skall kunna finna så många fel som möjligt i indata. Ett fel (en avvikelse från standard-Fortran) i en sats får ju inte förhindra programmet att analysera resten av satsen. Troligen krävs det tyvärr att användaren anger vilken dialekt han skrivit i för att programmet skall kunna finna de mera subtila avvikelser från standardfortran som även kunniga programmerare ofta gör.

6. Teori för programmeringsspråk

6a. SIMLISP

Introduktion och målsättning

Detta kapitel redogör för ett försök att använda SIMULA som ytspråk till ett LISP-system. Målsättningen är:

- att ge en formell beskrivning av SIMULA:s semantik
- att, med den formella beskrivningen, kunna exekvera SIMULA-program.

Som värdefulla biresultat kan nämnas:

- ett interaktivt SIMULA-system (om man har ett interaktivt LISP-system)
- Debugging-faciliteter under exekveringen, som t ex gå in och titta på variabelvärden· editera programmet, titta vilka procedurer som anropats m m.
- en definition av SIMULA, som också talar om vad som sker under kompilering, och vad som sker vid exekveringar.
- en typecheckningsrutin, som kontrollerar att programmets hantering av olika datatyper är korrekt. Kontrollen utföres före interpreteringen, och kan möjligen användas av ett kompilerande system, för att ta bort onödiga run-time-kontroller (som konventionell kompilor ej tar bort) eller för att på ett tidigt skede flagga för fel, som annars skulle inträffa under exekveringar.

Jämförelser med liknande arbeten.

Problemet "att ge en formell beskrivning av programmeringsspråk" har behandlats av en mängd personer på olika sätt. En översikt av olika kända 1968 ges av Baker [Ba]. I oktober 1973 gavs ett symposium som ägnades åt principer för programmeringsspråk [Pr]. I nedanstående litteraturförteckning används följande klassificiering:

- axiomatiska beskrivningar
- ej axiomatiska beskrivningar

Axiomatiska beskrivningar

(Alla datatyper i programmet beskrivs med axiom. Inference-regler för satser beskriver sedan hur programmets omgivning förändras vid satsens utförande). Tidigare arbeten är gjorde av:

Igarashi [Ig], Yanov [Ya], Luckham [Lu3]

Floyd [Fl], Burstall [Bu].

Det kanske mest intressanta arbetet är definitionen av programmeringsspråket PASCAL [Ho].

Ej axiomatiska beskrivningar

Den absolut vanligaste metoden här, är att ge beskrivningen i stil med

$$\xi' = I(P, \xi) \qquad P \in S$$

där ξ, ξ' är gamla resp nya tillståndet vid ett steg i exekveringen av programmet. P är ett program. I är en maskin. Beskrivningen svarar då, dels mot att beskriva maskinen I (som t ex en uppsättning evalueringsregler) dels att beskriva syntaxen för ett språk S. Man kan tillåta att S ej är det språk man tänker definiera (t ex SIMULA), om man även definierar översättningsfunktionen K i

$$P = K(p) \qquad p \in \text{det språk som skall definieras.}$$
$$P \in S$$

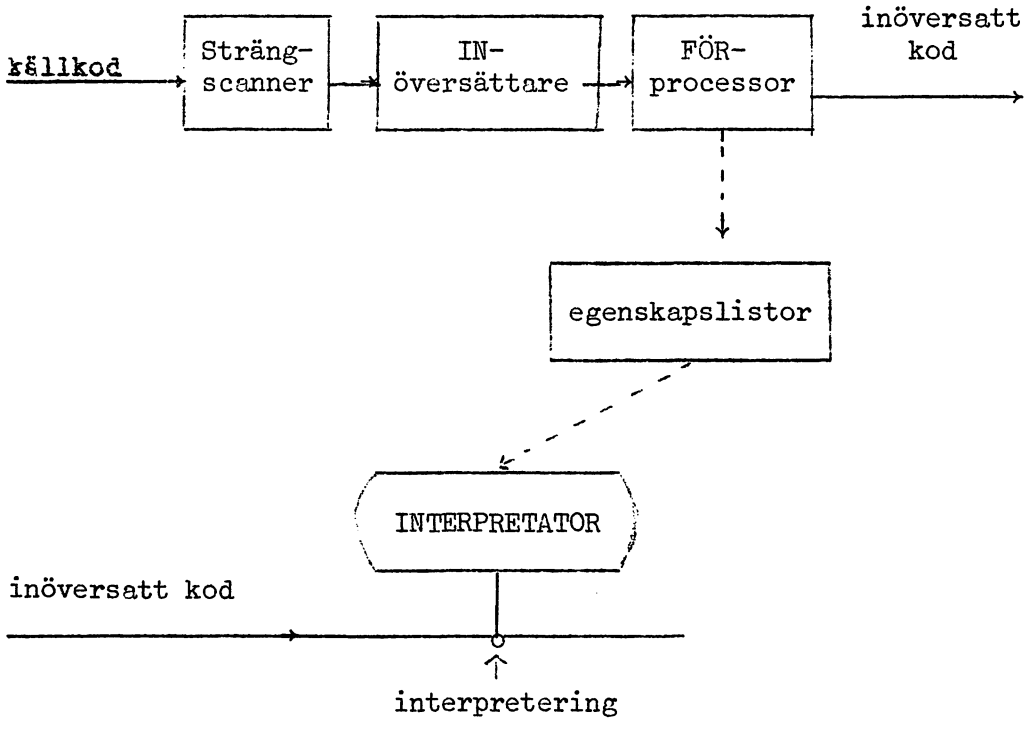
samt gramatiken för S. Infinitivt svarar K mot en kompilator och I mot en interpretator.

Ett tidigt arbete är gjort av McCarthy [Mc1]. Programmeringsspråket LISP [Mc2] använder sig av denna teknik. Det mest kända arbetet är VDL [Lu2], [We], (the Vienna Definition Language), som använts för att definiera semantiken för PL/1 [Lu2]. SIMULA-LISP projektet använder sig av idéer tagna bl a från VDL. Knuth [Ku] anger en metod att applicera funktioner till noderna i en ordinär BNF-grammatik. Dessa funktioner skall sedan användas för att ge den semantiska beskrivningen. Wilmer [Wi] och Fang [Fa] har applicerat denna idé på SIMULA. Deras funktioner är "kompileringsfunktioner" som översätter givet SIMULA-program till en serie Macros, som sedan interpreteras av en SIMULA-maskin.

Andra arbeten, som ej direkt berör den semantiska definitionen, men som är relevanta för detta projekt är:

- en modell för kontrollstrukturer i SIMULA (DAHL [Da]). Denna modell ligger till grund för den beskrivning som SIMULA-ALGOL projektet ger.
- typcheckning av Ledgard [Le2].
- andra metoder för implementering av komplicerade kontroll-strukturer. Här kan nämnas arbeten av:
  Dijkstra [Di] (kommenter om goto i Algol)
  Fenichel [Fc] (om implementering av läges-variabler).
  Leavenworth [Le1] (beskrivning av språket McG 360, ett språk gjort speciellt med tanke på att ge användaren tillgång till att definiera en egen kontroll-struktur i språket).

Översiktlig beskrivning av systemet



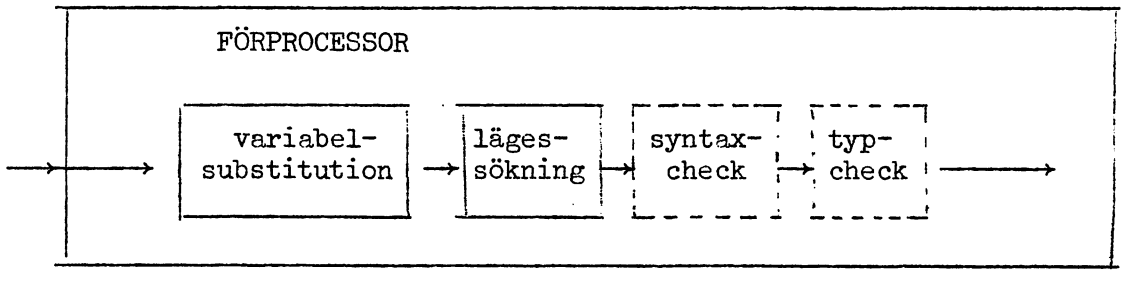


Fig 1.

Betrakta Fig 1.

Sträng-scannern läser ett SIMULA-element i taget
IN-översättare är f n en TOP-parser (se avsnitt 9c). Resultatet blir ett S-uttryck som nära motsvarar det inlästa programmet.

ex. begin real x,y;
x:=f(y)+x
end
översätts till
(BEGIN
(REAL X Y)
(:= X (+ (F Y) X)))

Förprocessorn: Översätter ovanstående typ av inöversatt kod till den representation som interpretatorn förutsätter, nämligen en samling objekt med namngivna attribut.

För att beskriva vilka olika objekt som motsvaras av olika SIMULA-konstruktioner används för närvarande en skissartad grammatik enligt nedanstående exempel:

(Attributvärden med stora bokstäver är konstanter, med små bokstäver är "variabler" eller namn på nya produktioner).

```
STATEMENT:
     STATEMENTTYPE     : BLOCK
     BLOCK NAME        :identifier
     DECLARATIONPART   : declarationlist
STATEMENT:
     STATEMENTTYPE     : ASSIGN
     LEFT              : expression
     RIGHT             : expression
DECLARATION LIST:
     DECLARATION       : declaration
     TAIL              : declarationlist
DECLARATION:
     DECLARATIONTYPE   : REAL | INTERGER | - - - - - -
     DECLARATIONVALUE  : declaration value
     IDENTIFIER        : atom
:
```

```
:
:
STATEMENT LIST:
     STATEMENT          : statement
     TAIL               : statementlist
```

Satsen begin real x,g;
x:=f(g)+x
end

får följande interna representation:

```
STATEMENT TYPE          : BLOCK
BLOCKNAME               : Bl
DECLARATIONPART         :
        DECLARATION     :
                        DECLARATION TYPE    : REAL
                        DECLARATIONVALUE    : 0.0
                        IDENTIFIER          : X
        TAIL            :
                        DECLARATION         :
                            DECLARATIONTYPE : REAL
                            DECLARATIONVALUE: 0.0
                            IDENTIFIER      : Y
                                              /* Observera, att TAIL
                                                 saknas här
STATEMENTPART:
     STATEMENT          :
                        STATEMENTTYPE       : ASSIGN
                        LEFT                :
                            BLOCKNAMN : Bl  /* "adressref" för variabeln
                            IDENTIFIER: X      X
                        RIGHT:              :
                            OP : +
                            ARG1 :
                                PROCEDURE REF :
                                    BLOCKNAME : BO
                                    IDENTIFIER: F
                                ARGLIST : ...
```

osv.

Ovanstående grammatik liknar den vanliga BNF-grammatiken och ges parallellt med denna i den fullständiga formella definitionen. (Fn undersöks, hur komplicerat det blir att definiera översättningen till objekt av ovanstående typ med hj. av Knuth:s metod [Kn].

Förprocessorn har ett antal fristående moduler för olika uppgifter, nämligen:

(a) identifierare-substitution

Vid varje förekomst av en identifierare kontrolleras i vilket block den är deklarerad. Identifieraren byts sedan ut mot ett objekt av typ BLOCKNAME : id
IDENTIFIER: id
(lägesidentifierare betraktas som deklarerade i det block identifieraren står som ett läge).

(b) läges-sökning

Till varje lägesidentifierare är associerad en "vägbeskrivning" hur man från blocket, där läget är "deklarerat" kommer till själva läget. Vägbeskrivningen används sedan av själva interpretorn. Detta åstakommes genom att pseudo-exekvera programmet (inga satsen utföres, men alla satser uppsöks). När ett läge påträffas, finns det i interpretatorn en kontrollstack som exakt motsvarar hur man kommer till läget. Denna kontrollstack ges som lägesvärde.

(c) syntax check

Utför kontroll av syntaxen. Denna modul kan tas bort (om man man t ex vet, att programmet är syntaktiskt korrekt.)

(d) Typ-check

(Utför den typ-checkning, som ej tas om hand av (c)). Kontrollerar, att alla satser opererar på tillåtna datatyper, samt indikerar, var sådan typcheckning ej kan utföras i förväg. I princip används samma interpretator som för den vanliga exekveringen. Alla identifierare och satsen m m returnerar dock en mängd av datatyper istället. Mer utförligt om denna rutin finns i kap 6a6 DLU -73. Även denna modul kan tas bort om man ej behöver den.

SIMULA-Interpretatorn

Interpretatorn för SIMULA är starkt influerad av VDL [We], utbyggd med en metod för beskrivning av co-rutinen given av Dahl [Da]. Här redogöres först för VDL-tekniken, sedan hur problemet med co-rutiner (detach och resume) har lösts.

VDL-tekniken

Utförligare beskrivning finns i DLU -73/4, [We]. Här följer en redogörelse för så mycket av VDL-språket som behövs för den fortsatta förståelsen. Ett steg S i en evaluering av en sats beskrivs i VDL av

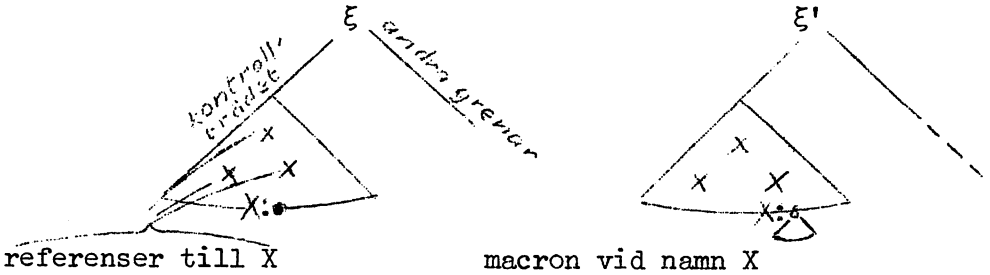
$$\xi' = I(s,\xi)$$

där ξ' är ett tillståndsträd som bär information, dels om programmets data, dels om kontrollstrukturen i programmet (dvs vilket steg som skall utföras härnäst). I är beskrivningen av språket (t ex SIMULA) given i VDL. Kontrollmekanismen ligger i en speciell "gren" i trädet ξ, och kallas i fortsättningen för kontroll-trädet. Noderna i kontroll-trädet kan uppfattas som ännu ej exekverade macro-anrop. Löven i kontroll-trädet är de macros som för tillfället står i tur att evalueras. (Vilket löv som evalueras först, om flera finnes, är odefinierat i VDL). Värdet av evalueringen av en macro definierar det nya kontroll-trädet. (Dessutom får macron ha bieffekter på andra grenar av tillståndsträdet ξ). Ett macro-värde kan vara:

(a) Ett nytt lokalt kontroll-träd. Detta lokala träd ersätter då macron, och trädet har expanderat.

(b) Ett sk pass-värde. Alla noder kan namnges. Dessa namn kan t ex användas som referens till noden från andra noder (oexpanderade macroanrop). Då pass-värde returneras; erhåller referensen motsvarande värde, coh själva noden försvinner.

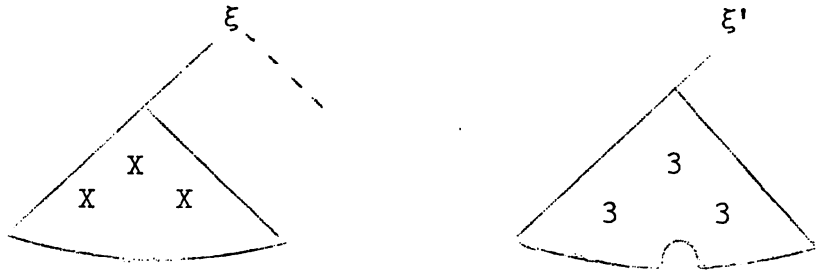
ex. Skiss av macroexpansion



referenser till X

macron vid namn X evalueras och returnar ett nytt lokalt träd.

Sliss av pass-värde



macron vid namn
X evalueras och returnerar
pass-värdet 3

Mer detaljerat exempel. Man kan tänkas definiera

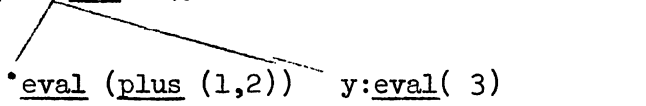
```
macro plus (a,b) :  add(x,y)          }   definierar ett nytt
                    y : eval (b)      }   lokalt träd. x och y är
                    x : eval (a)      }   nod-namn.
macro add  (x,g) :  pass (x + g)
macro eval (x)   :  if number (x) then pass (x)
                    else if macrop(x) then x
                    else ....
```

Vid evaluering av

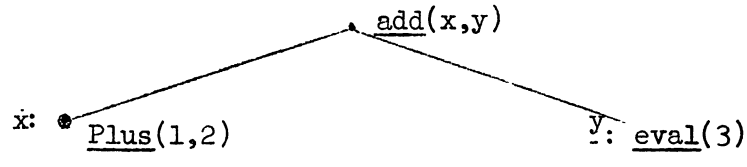
plus (plus (1, 2), plus (3, 4))

erhålles följande sekvens av träd:

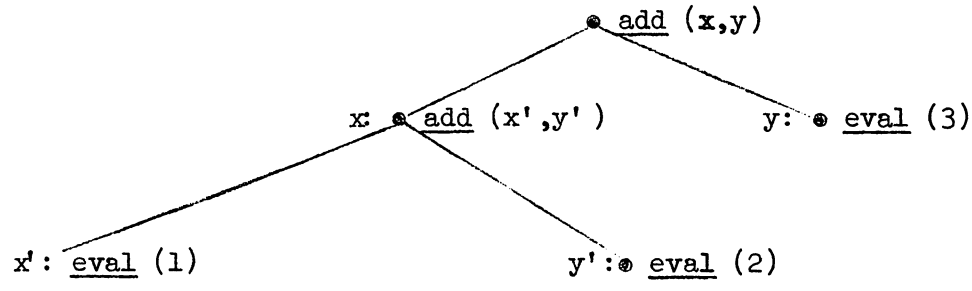
1. • plus (plus (1,2), 3)
2. • add (x,y)



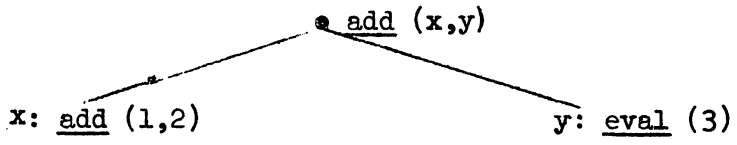
3.



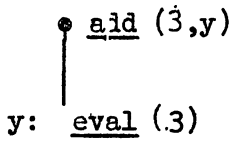
4.



5.-6. eval(1) resp eval(2) returneras 1, 2 som pass-värder till x' resp y'



7. add (1,2) returnerar 3 som pass-värde

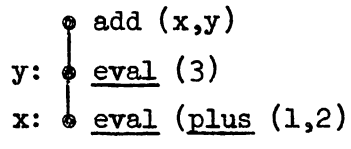


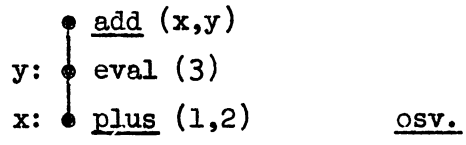
8. eval (3) returnerar 3 som pass-värde,

9. resultatet blir 6.

I SIMULA behövs ej möjligheten att samtidigt ha flera noder som kan expanderas, eftersom det är definierat, i vilken ordning all slags evaluering skall ske. Därför kan kontroll-trädet reduceras till en kontroll-lista, vars slutnod är den nod som skall expanderas. Vi får då en stack-mekanising med möjlighet att döpa stackelement vid namn och returnera värdet av dessa till andra stack-element. Vi kallar i fortsättningen kontrolllistan för kontroll-stacken. Ovanstående exempel blir då:

1. plus(plus(1,2),3)

2.


3.


Sekvensen

add (x,y)
y: eval (b)
x: eval (a)

i definitionen av plus förstås lättast som en bit kontrollstack med 3 element, som skall ersätta noden plus (-,-) vid expanderingen. Observera, att evalueringsordningen blir nerifrån och upp.

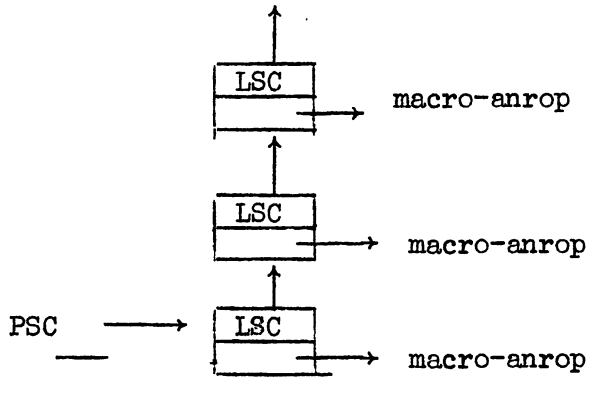
Tekniken för detach-resume

Hittills har vi beskrivit ett VDL-system. För att klara av detach-resume i SIMULA har två viktiga möjligheter tillförts VDL-språket.

(a) Möjlighet att lagra programmets alla variabelbindningar i själva kontroll-stacken.

(b) Möjlighet att manipulera med själva kontroll-stacken.

Att tillfälligtvis avbryta exekveringar vid någon punkt, för att senare kunna fortsätta svarar mot att lagra undan en del av kontrollstacken för att senare kunna "lägga tillbaka". Detta sköts, i enlighet med [Da] av två primitiver SWAP(X) och rotate(x,y) som definieras strax. För att notationen i SWAP och rotate skall förstås, visas först en skiss av kontroll-stacken:



PSC (Program Sequence Control) är en global VDL-variabel, som pekar ut nuvarande "botten" på kontroll-stacken (som växer uppifrån - ner). LSC står för Local Sequence Control (och kan ses som motsvarighet till återhoppsadresser vid funktions-anrop).

Definition av SWAP och Rotate:

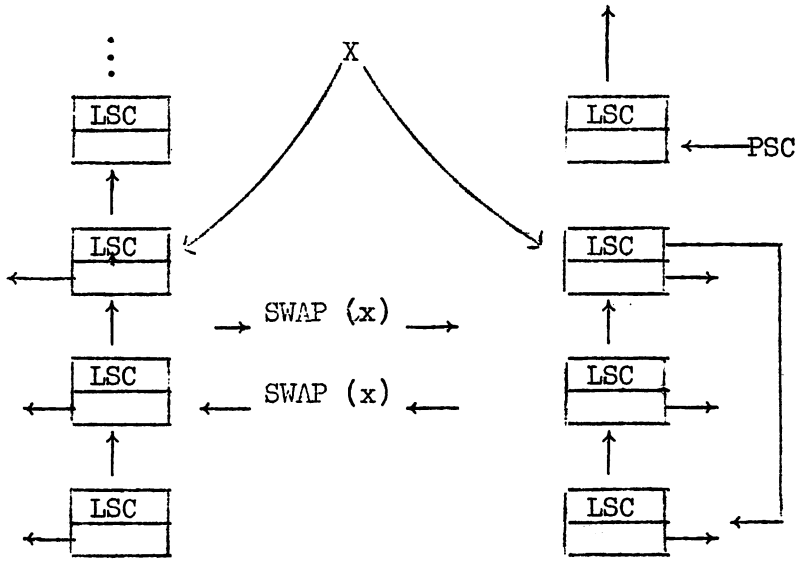
```
SWAP (x) = TEMP =psc;
           PSC: = x.LSC;
           x.LSC = temp;
```

Dvs innehållen i PSC och x.LSC byter plats.

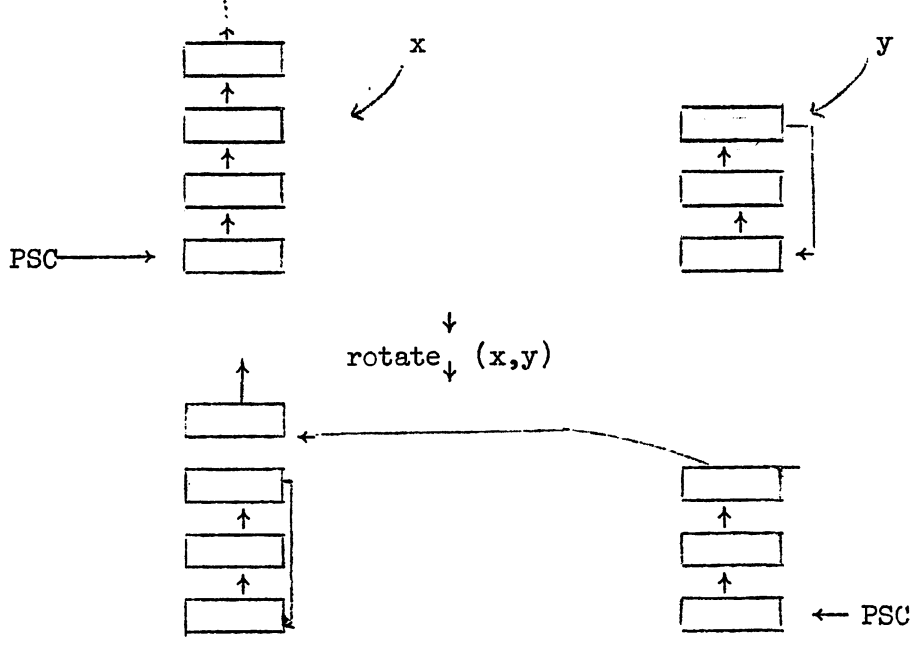
rotate (x,y) = SWAP (x); SWAP (y)

x och y är referenser till ett kontroll-stacks element.

Följande figur illustrerar SWAP (x)



Följande figur illustrerar rotate (x,y)



Nu kan detach, call och resume i SIMULA beskrivas med hjälp av SWAP och rotate enligt nedan. Vi antar att objekt som kan använda detach, call och resume, tillhör classen parproc.

```
class parproc;
      begin
          procedure detach;
              if this parproc ε OC then SWAP(this parproc)
              else error;
          procedure call (x); ref (parproc) x;
              if X ∉ OC then SWAP(x) else error;
          procedure resume (x); ref ('parproc) x;
              if X ε OC ∧ this parproc ∉ OC
                  then rotate( this parproc, X)
                  else error;
          end parproc;
```

OC (Operating Chain) = de "aktiva" blocken = de block som kan nås genom att följa LSC-pekare från PSC.

Utdrag ur den semantiska beskrivningen av SIMULA

Om notationen

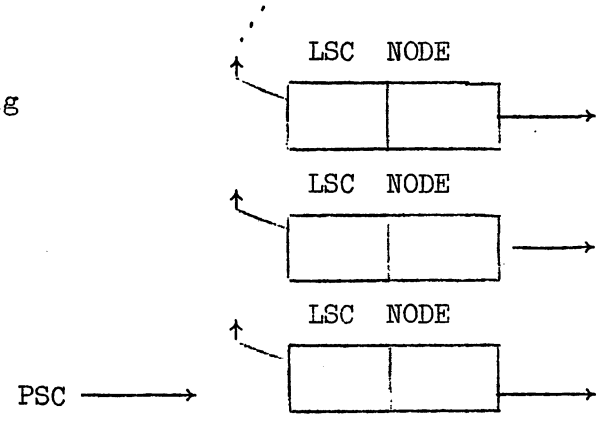
Alla macro-namn är understrukna. Övriga hjälpprocedurer är ej understrukna. Det är tillåtet att skriva if B then S, S_2 S_3 ... else S_2,. Dvs mellan then och else kan fler än en sak utföras. Ett macro-anrop skrivs enligt modellen

```
foo (argl, arg2 ...)
l1 : fij  (argl, ...)          rad 1
:
:
ln : fuu (argl, ...)           rad n
```

rad 1 - rad n är ej nödvändiga. Vidare behövs lägena l_i ej sättas ut om de ej behövs. Om makroanropet består av endast en macro, får [ ] uteslutas. Om värdet av en macroexpansion ej är ett macroanrop (enligt ovan) förutsätts pass-värde.

Den vanligaste datatypen är sammansatt objekt som är bärare av ett eller flera namngivna attribut. Objektets attribut kan nås via • -notation (jfr • -notationen i SIMULA). Kontroll-stacken har följande utseende:

fig



PSC är en "fri variabel" som alltid pekar ut nuvarande kontroll-stack. Stack-attributet NODE pekar ut resp stack-element.

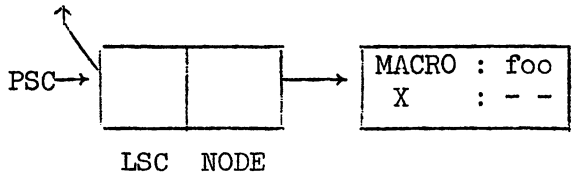
Representationen av namn-givning av noder i ett macro-anrop är oväsentlig för den semantiska beskrivningen och lämnas åt sidan. Ett macro-anrop foo (argl, ... arg_n) genererar ett objekt, som bl a har attribut = namnen på de formella parametrarna till foo. Små bokstäver indikerar variabler; stora bokstäver indikerar konstanter.

ex. foo (x) = [fii (1,2) ; fuu (3,4)]

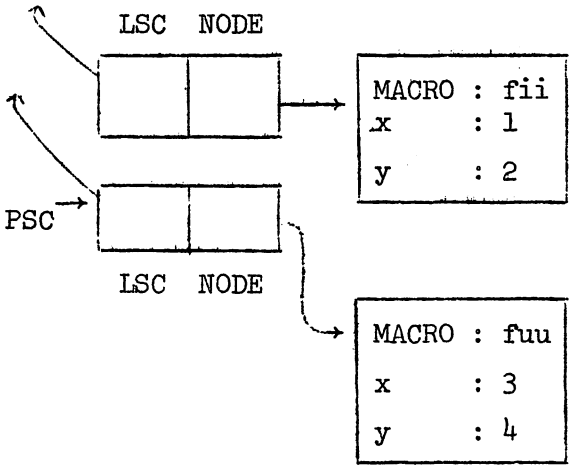
fii (x,y) = - - -

fuu (x,y) = - - -

ett anrop till foo är representerad genom strukturen



och generar strukturen



Hantering av variabelbindningar

En identifierares "adressreferens" karakteriseras av:

- sitt blocknamn (varje block har unikt namn)
- identifierarnamnet.

Vid förprocesseringen har alla förekomster av identifieraren ersatts med motsvarande adress-referens.

Vid interpreteringen evalueras en adress-referens enligt schemat:

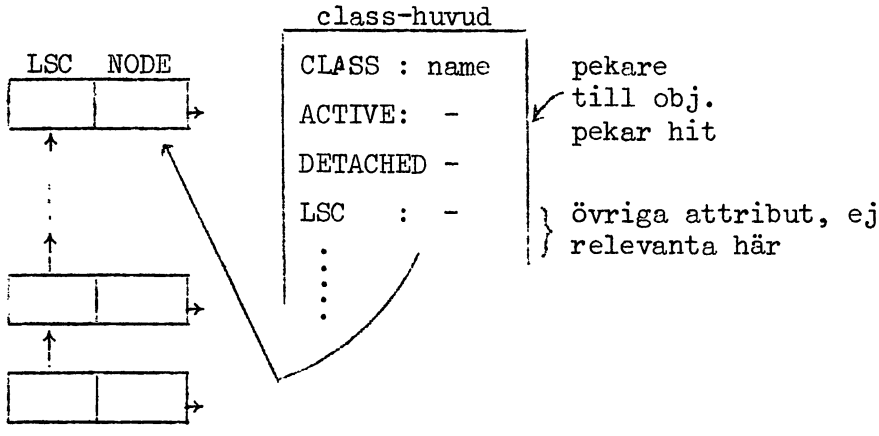
- sök första block med aktuella namnet
- sök identifierarnamnet i detta block.

Om adress-referenser svarar mot en name-deklarerad variabel som skall evalueras, läggs en pekare ut i stacken till den punkt i stacken som gällde, då procedure-anropet skedde. Vid sökning blir då genom denna pekare en bit av stacken tillfälligtvis dold.

Definition av SWAP och rotate (dessa är ej macros):

```
      SWAP(X) = TEMP := x. LSC;
      x. LSC := psc;
      PSC := temp;
      X;                        /* ge X som värde
rotate (x,y) = SWAP(x); SWAP(y);
```

Definition av Detach, call och resume:
En instans av ett objekt representeras av följande figur



detach och resume(x) i ett SIMULA-program har av en pre-processor översatts till detach (this (classnamn)) resp resume (this (classnamn), x)

this söker från PSC den första instansen av ett classhuvud med resp classnamn.

```
detach (x) = if is-active (x) then
    if is-classhead (x) then /* class objekt
        x. DETACHED = true;
        x. ACTIVE   = false;
        SWAP (x. LSC)
        else x    /* prefixed block
    else error

call (X) = if is-detachedandnotactive (x)
    then x. DETACHED = false
        x. ACTIVE = true
        SWAP (x. LSC)
    else error.

resume (x,y) = if is-detachedandnotactive (y)
    then
        if is-classhead(x)
            then rotate (x.LSC, y.LSC)
            else SWAP (y.LSC)
        else error
```

Definition av procedur och class-anrop

- procedure anrop :

```
apply_proc (name, argl) = [ blockhead (name, a)
                            evalbody (name. STM)
                            a: checkspec (b)
                            b: evalprocargs (name. FORMALPARAMS,
                                  argl, psc)
                          ]
```

evalprocargs returnerar en lista av par, där varje par svarar mot formell parameter-aktuell parameter. Aktuell parameter kan vara evaluerad (call by value) eller oevaluerad (call by name). I det senare fallet är den aktuella parametern kompletterad med stackpekaren psc vid anropstillfället. Denna pekare används för att hitta rätt omgivning, då argumentet senare skall avalueras.

check spec kollar, att argumenten svarar mot specifikationerna. Den parade listan ges sedan som pass-värde till blockhead.

eval body evaluerar proceduresatsen.

blockhead ett ex på databärande nod. Här har procedurens argument allokerats och erhållit värden.

class-anrop (new)

```
new (name, argl)=
    [ classhead (name, a, true, false)
      eval body (name.STM)
      a: addattributes (b, name. BLOCKATTRIBUTES)
      b: check spec (c)
      c: evalclassargs (name.FORMALATTRIBUTES, argl)
    ]
```

evalclassargs som evalprocargs fast call by name blir aldrig aktuellt.

addattributes kopplar ihop de två listorna.

classhead som blockhead, fast classhead märks med ACTIVE = true

Jämför f ö analogin med applyproc.

Anm en preprossessor har tidigare "slagit ihop" de block, som refereras via inner, samt lagt ut rätta namn-referenser vid virtuall-deklarationer. Detta är ex på sådant, som helt kan skötas av ett kompileringssteg.

Nuvarande status

Ett liknande system, men för ALGOL-60 är klart. Detta beskrivs i DLU -73. För SIMULA-systemet är hela INTERPRETATORN skriven. För närvarande håller FÖRPROCESSORN på att skrivas. I ett arbete från december 1972 [Fa] redogörs för en teknik att utnyttja BNF-grammatiken som en förprocessor till SIMULA, och för närvarande undersöks, vilket som verkar enklast av (a) att utnyttja denna teknik eller (b) att skriva ett konventionellt LISP-program.

Ett papper om en redogörelse för denna teknik jämfört med konventionell programmering är under utarbetande.

Referenser:

[Ba] Batcker, J. W., "Semantics of Programmering Languages". Advances in Information Systems Science, Vol 2, Chap 3.

[Bu] Burstal, "Formal Description of Program Structure and Semantics in First Orderlogic", Machine Intelligence 5, pp 79.

[Da] Dahl, O-J., Wang, A., "Co-routine Sequencing in a Block Structured Environment", BIT 11, pp 425.

[Di] Dijkstra, E., "Goto Statement Considered Harmful", CACM, mars 1968.

[Fa] Fang, I., "FOLDS" A Declarative Formal Language Definition System", Stanford University, Comp Science, nr 72-329.

[Fe] Fenichel, R., "On Implementation of Label Variables", CACM, maj 1971.

[Fl] Floyd, R., W., "Assigning Meanings to Programs", finns i, Papers in Theory of Computation.

[Ho] Hoare, C. A. R., Wirth, M. N., "An Axiomatic Definition of the Programming Language Pascal", Eidgenössische Teknische Hoch Schule, Zürich, nov 1972.

[Ig] Igirashi, S., "An Axiomatic Approach to the Equivalence Problems of Algorithms", PhD Thesis, Report of the Computer Center, University of Tokyo, 1-101 (68).

[Kn] Knuth, D. E., "Semantics of Context-free Languages", Mathematical System Theory, J. Z. Z, 1968.

[La] Lauer, P., "Formal definition of Algol 60", IBM Lab., Vienna, TR 25.088.

[Le1] Leavenworth, B. M., "The Definitions of Control structures in McG 360", IBM-research RC 2376, febr 1969.

[Lu1] Lucas, Lauer, Stigleitner, "Method and Notation for the Formal Definition of Programming Languages", IBM Lab., Vienna, Tr 25.087.

[Lu2] Lucas, P., "On the Formalization of Syntax and Semantics of PL/L", IBM Lab., Vienna, TR 25.060.

[Lu3] Luckham, D., "On formalized Computer Programs", Progr. Research Group, Oxford Univ, 1967.

[Mc1] McCarthy m m, "LISP 1.5 Programmers Manual", MIT-press.

[Mc2] McCarthy, J., "Towards a Mathematical Science of Computation", från Papers in Theory of Computation.

[Pr] Proceedings of ACM Symposium on Principles of Programming Languages, okt 1973.

[We] Wegner, P., "The Vienna Definition Language", ACM Computing Surveys, mars 1972.

[Wi] Wilmer, W., "Declarative Semantic Definition as Illustrated by a Definition of Simula", PhD, Stanford University, Computer Science, 1971.

[Ya] Yanov, Y. I., "The Logical Schemes of Algorithms", Problems of Cylemetics, Vol 1, pp 82-140, 1960.

6b. VDL-interpretator

6b1. Introduktion

VDL (the Vienna Definition Language) är ett metaspråk avsett framförallt för att ge formella semantiska beskrivningar åt programmeringsspråk.

VDL specificerades 1968 i IBM:s Wien-laboratorium,[1] och har använts bl a för att ge den formella definition för PL/L[2] och ALGOL 60[3]. En (lång) introduktion till VDL ges av P. Wegner[4].

Grundtanken bakom VDL är att alla operationer i det tänkta mål-språket (t ex ALGOL) skall kunna beskrivas som operationer verkande på ett tillstånds träd ξ, dvs att beskriva t ex ALGOL är equivalent med att ge en beskrivning I (i VDL) som definierar transformationen $\xi' = L(p, \xi)$, där p är godtyckligt ALGOL-program och ξ och ξ' gamla resp nya tillståndträdet efter ett steg i programexekveringen.

6b2. Målsättning och resultat av implementering av en VDL-interpretator i LISP

Under vårterminen 1973 skrevs i LISP en interpretator för metaspråket VDL. Givet en semantisk beskrivning i VDL för ett käll-språk, kan man använda denna interpretator + beskrivningen som en interpretator för källspråket i fråga.

Projektet har ingått som en del-studie i arbetet att ge en Semantisk definition av SIMULA -67 (se kap 6a). Målsättningen för delstudien har varit att undersöka möjligheterna att:

(a) använda VDL eller VDL-liknande metaspråk för definition av SIMULA 67;

(b) i så fall, kan man överföra en sådan definition till ett LISP-system, så att SIMULA -67 kan användas som ytspråk till LISP;

(c) hur svårt är det, att skriva en VDL-interpretator i LISP?

Resultatet av arbetet är beskrivet i detalj i ref 5. Här kan nämnas några synpunkter:

Svar på (a) efter att ha definierat om möjligheten att själv påverka kontrollstrukturer i VDL, anser jag att ett VDL-liknande metaspråk är väl lämpat för definition av SIMULA -67, se även kap 6a.

Svar på (c) VDL-interpretatorn består av C:a 200 kort LISP-kod och får ses som en lätt uppgift att skriva i LISP. Arbetet tog 2 veckor.

Svar på (b) Experiment med ALGOL-program har visat, att när man väl givit den semantiska beskrivningen, är det lätt att låta beskrivningen verka som interpretator för ALGOL-program. Beskrivningen förutsätter dock någon slags intern-representation av ALGOL-program, och de största svårigheterna ligger i att välja en intern representation så att:

(i) semantiska beskrivningen blir så lättläst som möjligt;

(ii) översättningen ALGOL-sträng till intern representation möjlig att koda (och beskriva!) så enkelt som möjligt.

Som biresultat från experimentet kan också nämnas:

- en semantisk beskrivning av VDL i VDL (något som saknas i VDL-rapporter[1]!)
- en semantisk beskrivning av LISP i VDL. Gjord mera för demonstrationsändamål.

6b3. Referenser

1. P. Lucas, P. Laner, H. stigleitner, "Method and Notation for the Formal Definition of Programming Languages", IBM Lab, Vienna, TR 25.087.
2. P. Lucas, "On the Formalization of Syntax and Semantics of PL/L", IBM Lab, Vienna, TR 25.060.
3. P. Laner, "Formal Definition of ALGOL -60", IBM Lab, Vienna, TR 25.088.
4. P. Wegner, "The Vienna Definition Language", ACM Computing Surveys March 72.
5. Detta papper - M. Nordström, "Synpunkter på VDL sett från LISP DLU 73/4.

PROGRAMREDSKAP OCH METODER

7. Bakgrund och översikt

En målsättning vid Datalogilaboratoriet är att bygga upp en "redskapslåda", ett bibliotek av smådatabasprogram som kan komma till användning i olika tillämpningar. Detta mål förutsätter bland annat att man har ett gemensamt programmeringsspråk att arbeta i. Vi har valt en LISP-dialekt, dvs INTERLISP (f d BBN-LISP) som denna gemensamma bas.

Under 1973 har situationen komplicerats av att vi haft tillgång dels till Uppsala Datacentrals maskin, där det nyutvecklade INTERLISP-systemet använts jämsides med det gamla LISP F1, dels till DEC 1070-systemet ("Simon") vid Stockholms Datamaskincentral. Även för den senare finns ett INTERLISP-system, men det kan praktiskt användas först när man infört det nya operativsystem som understöder virtuellt minne. Med nuvarande operativsystem kan endast Stanford-LISP för PDP-10 köras. Samtidigt har DEC-systemet ur vissa synpunkter (interaktivt tillgänglig hela dagen, mm) för många projekt varit mer attraktivt att köra på än UDAC:s 370/155.

Vi har därför under året fortlöpande haft kompatibilitetsproblem mellan tre olika LISP-dialekter. Detta har gått att klara genom automatiska översättare som täcker de flesta fall, defensiv programmering, etc, men har ändå krävt visst arbete.

Härigenom har också arbetet på ett gemensamt programbibliotek något försvårats. Vi har i princip siktat mot användning av INTERLISP, och därför hållit igen på användningen av Simon för att undvika alltför mycket framtida konverteringsarbete, men ändå finns vissa program på Simon eller LISP F1 men ej under INTERLISP.

Följande är en lista på de paket som f n (januari 1974) helt inarbetats i INTERLISP-programbiblioteket, dokumenterats, etc:

programnamn	användning	se avsnitt
GIP/GUP	in- och utmatning av databasdelar	
MAKRO	makroexpansion i program lagrade som liststrukturer	
MAP	hjälpfunktioner för vissa looptyper	
NWC NWP	nätverksparser ("kompilator" resp "interpretator" för grammatik)	
PCDB	predikatkalkyldatabas	
PMG	programmanipulering	
REDCOMPILE REDFUN	programoptimering	
REMREC	rekursionsborttagning	
SP	hjälpfunktioner på lågnivå	

Ytterligare ett antal program som beskrivs i denna årsrapport är skrivna och uttestade i INTERLISP, och är på väg in i biblioteket. Under vårterminen planeras en doktorandkurs, där deltagarna får köra de olika programmen i INTERLISP-biblioteket.

Under året har programsammankopplingar gjorts vid flera tillfällen:

projekt	använda hjälpprogram	se avsnitt
PLAST	IKP och TOP	9c
SIMLISP	" "	9c
SNVDATA	IKP	9c
Nat.språk	Nätverksparser + SEPAK	10b, 10c

Sammanfattningsvis har programbiblioteket utnyttjats för kommunikation med användaren, snarare än för sökning o a manipulation av databasen. Vi väntar oss dock att hjälpprogram även för den senare typen av användning kommer till användning så småningom. - Vidare har hjälpprogram för programhantering, ss REDFUN, MAKRO, etc använts regelbundet

för att operera på andra program. Sammankopplingarna har i stort sett gått bra, även om de har markerat önskvärdheten av väldefinierad programfunktion (något som inte alltid gäller för program som utvecklats som forskningsprojekt, där alltså programmet specificeras allteftersom arbetet pågår). Vi tycker därför att idén med en redskapslåda av hjälpprogram successivt håller på att förverkligas.

I föregående årsrapport gavs (i det kapitel som motsvarar detta) ett försök till systematisering av de önskvärda programredskapen i två dimensioner: programfunktion och datanivå (hos de data som programmet bearbetar. Nivån går på en skala från maskinnära till tillämpningsnära). Denna uppdelning synes fortfarande adekvat, och användes för att strukturera programbiblioteket.

8. Manipulation av LISP-program

8a. REDFUN och REDCOMPILE

En av huvudmålsättningarna för arbetet vid Datalogilaboratoriet är att framställa programredskap för användning i olika smådatabastillämpningar. Sådana programredskap kan i princip skrivas antingen som programgeneratorer (vilka genererar ett skräddarsytt program för varje tillämpning) eller som generella, parameterstyrda program. De förra har fördelen att de kan bli betydligt effektivare (eftersom datastrukturer kan skräddarsys, och eftersom man slipper ligga och testa på parametrar hela tiden). Generatorerna har dock samtidigt nackdelen att de är mer besvärliga att skriva än de generella programmen.

Vid DLU framkom tidigt idén att lösa denna motsättning genom att man, för varje önskat bidrag till programbiblioteket, först skriver ett parameterstyrt program, P, och därefter för varje tillämpning ger P + de önskade parametrarna, R, till ett optimeringsprogram, O, vilket sätter in parametrar och därefter utför möjliga förenklingar. Om det genererade programmet kallas P'_R, och dess indata I, har vi alltså

$$P'_R = O(P,R)$$

$$P(R,I) = P'_R(I)$$

Under uttestningen av P kan man ev föredra att göra körningar av typen P(R,I), eftersom det troligen inte lönar sig att göra om optimeringen för varje bug man rättar, men så snart P är färdigt och skall användas, utföres optimeringen för varje tillämpning.

Denna metodik kom först till användning inom PCDB-programmet, och har beskrivits bl a i tidigare årsrapporter. Jämsides med PCDB utvecklades programmet REDFUN, vilket skulle tjäna som optimeraren O ovan. I fallet med PCDB skrevs programmet P på ett sådant sätt att det skulle passa att optimera med REDFUN. Detta innebar bl a att programmet skrivs helt procedurellt, och att inga tilldelningssatser, goto-satser, el dyl förekom i P.

Vi ville därefter undersöka vad som händer om man skriver P helt fritt som ett parameterstyrt program, utan hänsyn till att det senare skall optimeras. Man kan ju förvänta att detta ställer större krav på optimeraren O. För experiment med detta valdes problemet med inläsning och presentation av delar av en smådatabas. Parameterstyrda program för detta, kallade GIP resp GUP, skrevs under våren 1973, och beskrivs i avsnitt 9b nedan. Under hösten 1973 gjordes så experiment med att specialisera dessa medelst REDFUN. Följande slutsatser kunde dras:

a) GIP/GUP bestod av ett jämförelsevis stort antal funktioner, men endast en mindre del av dessa behövde optimeras. Övriga, mindre frekventa eller mindre parameterberoende procedurer kunde kvarstå i sin generella form.

b) De procedurer som borde optimeras hade "spontant" programmerats med användning av tilldelningar till slaskvariabler, m fl egenskaper som REDFUN ej klarade. Därför erhölls endast en försumbar förbättring vid optimeringsförsöken.

c) Man kunde ganska lätt se vilka ytterligare programoptimeringsmetoder som borde byggas in i REDFUN för att nästan all intuitivt möjlig optimering verkligen skulle utföras. Vid experiment där optimering enligt dessa regler utfördes för hand, erhölls en tidsvinst på uppemot 50% i gynnsamma fall (för interpreterad kod).

Nästa steg blev därför att införa dessa förbättringar i REDFUN. Arbete härpå pågår f n. Målsättningen är att utveckla REDFUN så att det klarar att optimera GIP/GUP tillfredsställande, och därefter att prova det på fler exempel, samt också att GIP/GUP efter optimering blir så effektivt att det blir praktiskt användbart.

Under 1973 utfördes vidare arbete på en "kompilator" för REDFUN. Idén är följande: I vissa fall vill man utföra genereringen av P' = O(P,R) ett mycket stort antal gånger, för samma P men olika R. Programmet O är av flera skäl ganska långsamt. Man skulle då i princip vilja specialisera O, dvs konstruera

$$O'_P = O(O,P)$$

och sedan vid varje genereringstillfälle utföra

$$O'_P(R)$$

vilket ju enligt ovan skall ge samma värde som P'_R som $O(P,R)$.* Detta sätt att konstruera O' förutsätter dock att O är tillräckligt rent skriven, eller kraftfull, för att kunna specialisera sig själv. Det villkoret var ej uppfyllt för REDFUN.

En möjlighet var då att "rensa" i koden för O. Vi valde dock en annan väg, som bedömdes som enklare, nämligen att skriva ett program C som genererar ett program C(P) som med R som argument genererar

$$C(P)(R) = C\Psi\, P'_R$$

C(P) är alltså ekvivalent med det ovan önskade $O'_P = O(O,P)$. Programmet C är då det vi kallar REDFUN-kompilatorn, REDCOMPILE.

Programmet REDCOMPILE skrevs under sommaren 1973 med målsättningen att vara kompatibel med den då existerande varianten av REDFUN. Detta mål uppnåddes också i stort sett. De modifikationer i REDFUN som sedan dess införts, har ännu ej införts på motsvarande sätt i REDCOMPILE.

8b. REMREC: prestandamätningar

Under 1972 skrevs ett LISP-program, "REMREC", för automatisk rekursionsborttagning i LISP-funktioner. (Se DLU:s årsrapport 1973). Eftersom INTERLISP-kompilatorn nu fungerar har prestandsmätningar kunnat göras på REMREC

---

* Detta bevisas genom att i de ovan givna likheterna substituera P↦O, R↦P, och I↦R, varvid erhålles

$$P'_P = O(O,P)$$

$$O(P,R) = O'_P(R)$$

samtidigt som vi direkt hade $P'_R = O(P,R) = O'_P(R)$

Ett antal tester är utförda för att jämföra exekveringstider för vissa enkla LISP-funktioner före och efter applicerande av REMREC. Jämförelse har också skett mellan kompilerad kod och okompilerad sådan. De funktioner som testats kan anses utgöra ett slags grundfunktioner för resp "rekursionstyp" (se DLU 1973 och/eller dok. av REMREC). Tiderna som erhållits torde därför utvisa optimeringsgraden för REMREC i fördelaktiga fall. Nedan följer först testfunktionernas utseende före och efter REMREC:s applicerande. Därefter kommer tablåer över körtider m m. Testfuntioner (siffrorna inom parentes hänvisar till resp tablå i nästa kap).

Typ RCONS (1)

Ursprunglig kod:

```
copyl (x) =: if null(x) then nil
   else cons(car(x),copyl(cdr(x)));
```

Efter REMREC:

```
copyl(x) == prog( (B E)
            B:=E:=cons(NIL,NIL);
      copyl: if null(x) then begin cdr(E):=NIL
                                   return(cdr(B)); end
            else begin cdr(E):=cons(car(x),NIL);
                       x:=cdr(x);
                       goto copyl; end; );
```

Typ RPLUS (2)

Ursprunglig kod:

```
length(x) == if null(x) then 0
             else addl(length(cdr (x)));
```

Efter REMREC:

```
length(x) == prog ( (sum)
             SUM:=0
       length: if null(x) then return(sum)
             else begin SUM:=addl(SUM); X:=cdr(x);
                        goto length; end; );
```

Typ RNCONC (3)

Ursprunglig kod:

```
rev(x) ==if null(x) then NIL
          else nconcl(rev(cdr(x)),car(x))
```

Efter REMREC:

```
rev(x) == prog ( (B)
     rev: if null(x) then return(B)
          else begin B:=cons(car(x),B); x:=cdr(x);
                goto rev; end)
```

Typ RNCONC (4)

Ursprunglig kod:

```
rev(x) == if null(x) then NIL
          else nconc(rev(cdr(x)),cons(car(x)))
```

Efter REMREC:

```
rev(x) == prog ( (B)
     rev: if null(x) then return(B)
          else begin b:=nconc(cons(car(x)),L9;
                    x:=cdr(x); goto rev; end )
```

Typ RFOA (5)

Ursprunglig kod:

```
length(x) == if null(x) then 0
             else adda(length(cdr(x)))
```

Efter REMREC:

```
length(x) == prog((B E)
             E:=0;
      length: if null(x) then begin B:=0; goto L2; end
             else begin E:=addl(E); goto length; end;
          L2: if zerop(E) then return(B)
             else begin B:=adda(B); E:=subl(E); goto L2; end)
```

Def. av adda:

```
adda(x) == addl(x)
```

Körtider

Varje funktion har testats på 4 olika sätt. (Före och efter kompilering resp. före och efter REMREC) Vid varje test har funktionen ifråga tidstestats vanligtvis 5 varv i en loop. Detta har upprepats 3 ggr. De nedan angivna tiderna utgör medelvärdet av de 3 testerna. (Detta förfarande har tillämpats för att få en uppfattning av felet i tidmätningsfunktionen). Standardavikelse anges inom parentes.

(1)	INTERPRETERAT	KOMPILERAT
Före RR:	2696 (19)	190 (7)
Efter RR:	2372 (35)	117 (3)
Tidsbesparing:	12%	38%
(2)		
Före RR	5196 (62)	293 (14)
Efter RR	3527 (132)	96 (4)
Tidsbesparing	32%	67%
(3)		
Före RR	3098 (18)	650 (12)
Efter RR	1974 (74)	96 (13)
Tidsbesparing	36%	85%
(4)		
Före RR	3068 (79)	499 (6)
Efter RR	2259 (34)	252 (8)
Tidsbesparing	26%	49%
(5)		
Före RR	5091 (162)	592 (28)
Efter RR	7297 (126)	434 (15)
Tidsbesparing	-43%	26%

8c. MAKRO

Inledning

Vid övergången från det gamla FORTRAN-LISP-systemet till INTERLISP-systemet uppstod behov av något slags översättningsprogram. De vanligast förekommande skillnaderna mellan LISP-system är:

1. Ordningen på vissa funktioners argument är omkastad;
2. Funktioner har annorlunda namn.

För att lösa dessa båda problem utvecklades ett makro-expanderingssystem kallat MAKROEXPAND. Detta paket är dessutom mycket användbart för editering i LISP i allmänhet.

I systemet ingår två sorters MAKRO:n: "MAKROn" och "EMAKROn". De funktioner vars anrop man vill förändra (+ vissa FEXPR-funktioner) skall ha en (E)MAKRO-definition, som beskriver ändringen. När så alla (E)MAKROn är definerade applicerar man funktionen MAKROEXPAND på de funktioner man vill editera i.

(E)MAKROn utgöres av funktionsuttryck nedlagda på resp makrobärares egenskapslista.

MAKROn

Varje FEXPR-funktion, som ej evaluerar sina argument "normalt" (dvs p.s.s. som EXPR-funktionerna) skall ha en MAKRO-definition. Exempel på sådana funktioner är PROG, SELECTQ och COND. Denna MAKRO har till uppgift att:

1. Beskriva hur evalueringen av argumenten till funktionen skall ske;
2. I vissa fall ändring av argumentstrukturen (omkastning etc).

Värdet från evalueringen av en "MAKRO" skall vara en form ekvivalent med aktuellt funktionsanrop. Om värdet är NIL är effekten densamma som om det ursprungliga funktionsanropet returnerats. Ingen ytterligare genomgång av funktionsanropet.

Variabellistan i en MAKRO binds med hjälp av APPLY till argumenten i aktuellt funktionsanrop.

```
Ex  Antag att FOO  i
          (FOO A B C)
    har MAKRO:n
          (LAMBDA (X Y Z) ..... ).
    Då binds
          X till A
          Y till B
          Z till C.
```

På samma sätt om FOO :: MAKRO = (LAMBDA L ....)
binds

```
        L till (A B C).
```

Uttrycket (LAMBDA variabellista ....) kan läggas ned på FOO:s egenskapslista under indikatorn MAKRO med hjälp av funktionen MAKRO:

```
        (MAKRO FOO variabellista .... ).
```

För att beskriva hur aktuellt PMP skall fortsätta behandlingen av sub-

uttryck i aktuellt funktionsanrop då egenskapen MAKRO finns utlagd på den anropade funktionens egenskapslista är de fria variablerna MAKEVAL, MAKEVLIS och MAKFUN bundna till funktioner som skall appliceras på former, argumentlistor resp funktionsuttryck.

Ex. (MAKRO COND L (CONS 'COND (MAPCAR L MAKEVLIS)))
(MAKRO FUNKTION (X) (LIST 'FUNCTION (APPLY$_{*}$ MAKFUN X)))

Observera att man ej behöver applicera MAKFUN på car av aktuellt funktionsanrop ty detta sköter aktuellt PMP om. (Man behöver t ex ej applicera (APPLY$_{*}$ MAKFUN 'COND) ovan fastän COND ju är ett funktionsuttryck). Standardmakron för grundfunktioner som COND, SELECTQ osv ingår i systemet.

EMAKROn

EMAKROn skiljer sig från MAKROn genom att argumenten till "variabellista"redan är expanderade då funktionsuttrycket appliceras. Returneras värdet NIL från en EMAKRO är effekten densamma som om ingen (E)MAKRO funnits definerad. EMAKROn har sin största användning vid namnbyten och argumentomkastningar. EMAKROn läggs ut pss som MAKROn med hjälp av funktionen EMAKRO.

Ex. Nedan följer några exempel på bruk av MAKROEXPAND. Exemplen är realistiska vid översättning från INTERLISP till LISP 1.5. Nedan betecknar x,y,z i en form godtyckliga subuttryck.
→ = "skall överföras till"

(1) (PUT x y z) → (PUT x z y)

```
EMAKRO:
    (EMAKRO PUT (x y z)
          (LIST 'PUT x z y))
```

(2) (GET .....) → (GETP ....)

```
EMAKRO:
    (EMAKRO GET L (CONS 'GETP λ))
```

(3) (LIST X) → (CONS X NIL)

(LIST x_1.... x_*) → (LIST x_1 ... x_*) om K>1.

```
EMAKRO:n
    (EMAKRO LIST L
          (COND ((cdr L) NIL)
                (T (LIST 'CONS (car L) NIL))))
```

8d. PMG - En Program-Manipulator-Generator

Inledning. Vid skrivande av programmanipulerande program (PMP) i LISP behöver man i regel följande centrala algoritmer:

1a. En algoritm som genomsöker koden ungefär som en interpretator.

1b. En mekanism för behandlande av anrop till icke-evaluerande funktioner (FEXPRar).

Problemet vid 1b. löses i regel på så sätt att man lägger ut kod under någon indikator på resp funktions egenskapslista. Denna kod används sedan för att:

2a. beskriva hur evalueringen av delstrukturer i argumenten skall ske.

och/eller

2b. ersätta FEXPR-funktionsanropet med modifierad kod. (T ex förenkling). Detta är en form av makroexpandering.

Exempel på kod av typ 2a. är SPECIAL-egenskaperna i programmet FUNCSTRUC av Mats Nordström. Exempel på kod av både typ 2a. och 2b. är REDUCER-egenskaperna i REDFUN. (Se dessa båda programs dokumentationer.)

Det exakta utseendet på den utlagda koden varierar från program till program. Detta förhållande resulterar i att man måste skriva speciell kod av typ 2a. eller 2b. för varje nytt PMP man önskar använda. Det vore därför önskvärt om man kunde standardisera den utlagda koden 1b. på något sätt.

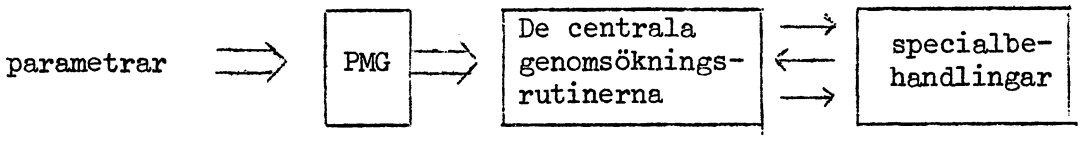
PMG

Jag har tidigare föreslagit att makrokonventionerna i paketet MAKROEXPAND (1) skall användas som en standard för kod av typ 2.
PMG är ett program som genererar centrala genomsökningsrutiner vilka tillämpar ovanstående makrokonventioner. PMG genererar alltså algoritmerna 1a. och 1b. Den utlagda koden kallas i fortsättningen makron. Det är relativt lätt att ändra i PMG så att det tillämpar andra makrokonventioner. PMG genererar 3 rutiner (kallade -eval, -evlis, -evfun). Till dessa centrala rutiner kan sedan egen specialbehandling av t ex variabler och funktionsuttryck anslutas. Man behöver därvid inte bekymra sig om hur genomgången av den av sitt PMP bearbetade koden skall ske.

PMG har bl a används för att generera följande PMP:

MAKROEXPAND	
RECURSIVEP	testar om funktion är rekursiv
FREEVARS	tar fram fria variabler
FUNCSTRUC	
RUV	tar bort oanvända variabler
CALLSTRUC	skriver ut anropsstruktur.

Fig. ⟹ dataflöde ⟶ kontrollflöde



Specialmakron

Många PMP fordrar specialbehandling av vissa funktioner. För sådana PMP tillåter PMG att man har funktionsuttryck utlagda under andra indikatorer än MAKRO. Dessa funktionsuttryck kallar jag SMAKROn. SMAKROn behandlas med högre prioritet än MAKROn. Det är även möjligt att specialbehandla funktioner genom att lägga in specialtester direkt i den av PMG genererade koden (IMAKROn). En vanlig typ av specialmakron är EMAKROn. (Se 8c).

Anrop av PMG

Programgenereringen utlöses av funktionen

PMG (SYS_* $PARMS_*$) FEXPR

där

SYS_* är ett atomiskt namn på det PMP som PMG skall generera.

$PARMS_*$ är en parameterlista som styr genereringen. $PARMS_*$ utgöres av en associationslista, vars sublistor har utseendet: (parameter v_1 ... v_k), där v_1 ... v_k bestämmer värdet på "parameter" och de åtgärder som skall vidtagas i de olika fallen. "parameter" kallas i fortsättningen subparameter till skillnad från SYS_* och $PARMS_*$:

Värdet från PMG är en lista över alla genererade funktioner plus de

funktioner som PMG väntar sig att användaren skall definera. Denna lista är mycket användbar då man senare vill göra en MAKEFILE på sitt PMP.

Exempel

1. Ett PMP som expanderar MAKROn, EMAKROn och inget mer erhålls med

```
(PMG Mx ((EMAKRO T) (REBINO T)))
(DE Mx (F)
    (PUTD f (MX-EVAL (GETD F))))
```

(RCBIND T) anger att PMG skall generera bindning av MAKEVAL, MAKEVLIS och MAKFUN.l

PMG genererar här funktionerna

MX-EVAL som appliceras på former

MX-EVLIS som appliceras på argumentlistor

MX-EVFUN som appliceras på funktionsuttryck.

2. Funktionen RECURSIVEP (f) som ger värdet T om f är rekursiv och NIL annars kan erhållas med

```
(PMG RP ((FUNCTION T) (REBIND T)))
(DE RP-FUNCTION (f)
    (COND ((EQ F FOO) (SETQ RES T))))
(DE RECURSIVEP (foo) (PROG (RES)
    (RP-evfun (GETD foo))
   (return RES)))
```

(FUNCTION T) anger att funktionssymboler skall specialbehandlas. PMG genererar då anrop till RP-function, som användaren får definiera.

PMG genererar här funktionerna RP-EVAL, RP-EVLIS och RP-EVFUN.

8e. Funktionslexikon

Under en interaktiv session (resp en batch-körning) med ett LISP-system ligger funktionsdefinitioner (= procedurer) lagrade som liststrukturer. Mellan körtillfällena kan dessa sparas antingen genom att hela datastrukturen bibehålles, eller genom att definitionen skrives ut som text (med utnyttjande av systemets "print av datastruktur") på separata filer.

I båda fallen är det emellertid önskevärt att definiera och namnge mängder av sammanhörande funktioner, varvid man alltså kan utföra kommandon ss.

"spara undan funktionsgruppen SEARCHFNS"

" gör en snygg utskrift av funktionsgruppen PARSER"

"lägg till funktionen TRANSITIVE till funktionsgruppen SEARCHFNS"

Ett större LISP-program (t ex PCDB som består av ca 375 funktioner) uppdelas alltså på ett antal funktionsgrupper. Dessa kallas något oegentligt för filer.

För att hjälpa en användare att hålla reda på sitt funktionsbibliotek har programmet FILEINDEX framställts. Programmet merg r funktionsnamn från olika filer till en alfabetiskt ordnad lista. Funktionsnamnen skrivs sedan ut i en tabell, där det också anges på vilken fil funktionen återfinns. Förhoppningsvis hjälper denna en användare att överblicka sin produktion.

Jämsides med funktionsnamn behandlas även (såväl i "filerna" som i programmet FILEINDEX) globala variabler och egenskapslistebärare.

9. Programredskap för informationsstrukturer på lägre nivå

9a PCDB - Predicate Calculus Data Base

PCDB-paketet förutsätter att användaren beskriver problemomgivningen i predikatkalkyl. Han skall deklera de relationer och funktioner han behöver, och vilka axiom som gäller för dessa. PCDB-paketet består av en programgenerator, som från dessa deklarationer och axiom genererar rutiner för lagring, hämtning, radering och slutsatsdragning av formler i denna kalkyl, samt av ett "run-time system" av hjälpfunktioner som anropas av den genererade koden.

PCDB har beskrivits ganska utförligt i de föregående årens rapporter (DLU-71, DLU-72 och DLU-73) och inriktningen på arbetet har sedan dess inte ändrats. Den intresserade läsaren hänvisas därför till dessa årsrapporter. En kortfattad beskrivning av PCDB finns emellertid i en ny rapport (ref 1).

Arbetet på PCDB har fortgått enligt följande: En första version (A-versionen) skrevs under 1971 och våren 1972, dokumenterades (ref 2) och användes för tillämpningsexperiment. Detta arbete utfördes i huvudsak av Lennart Drugge. En ny kompilator (B-versionen), som direkt genererar den slutgiltliga koden utan några optimeringspass, skrevs sedan av Rene Reboh. Under hösten 1972 påbörjades arbetet med en ny reviderad version (C-versionen) och detta arbete har fortsatt under 1973. Detta arbete har utförts av Anders Haraldson. Denna C-version kommer att "låsas" och dokumentation kommer att utarbetas. Del I av systemdokumentationen skrevs under hösten 1972 och den återstående delen innehållande axiomkompilatorn beräknas att bli klar under våren 1974.

Arbetet under 1973 på PCDB har bestått av följande delar:

- Axiomkompilatorn har helt omarbetats och har implementerats på LISP F1.
- Infört möjligheten att ta bort fakta ur databasen. Detta kan ske antingen med funktionen remove, som tar bort explicit lagrad fakta eller med funktionen erase, som även tar bort implicit lagrad fakta beskriven av axiom.
- Konvertertering av hela systemet från LISP F1 till INTERLISP.

Axiomkompilatorn. Låt oss först se i vilka olika utföranden ett axiom kan förekomma i PCDB och sedan följa ett exempel.

Assert Då ett påsteående lagras anger axiomet att ytterligare påståenden skall lagras. Exempel: Om "x är större än y" skall lagras vill vi även att "y är mindre än x" skall lagras.

Erase Då ett påstående raderas ur databasen anger axiomet att även andra påståenden skall raderas. Rena motsatsen till assert.

Search/Recsearch Används för att besvara slutna frågor, på fakta som ej är explicit lagrat. Denna sökning kan antingen utföras bredden först eller djupet först (rekursivt).

Find/Recfind Som föregående men för att besvara öppna frågor.

Ovanstående funtionsnamn används vid kommunikation med PCDB. Internt används motsvarande sys-funktion. Assert har alltså en motsvarighet i sys-assert.

Vid användandet av axiom för assert och erase utförs sk framåtslutsatsdragning och i övriga fall sk bakåtslutsatsdragning.

Låt oss följa axiomet

((FAR X Y)(BROR Z Y)(SON Z X))

med tolkningen "om x är far till y och z är bror till y gäller att z är son till x" och se vilken LISP-kod, som genereras för axiomet i några olika utföranden. Tre relationer far, bror och son deklareras som relationer, axiomet lagras och kallas för testex. Sedan kompileras axiomet och den genererade koden skrivs ut. Som avslutning visas några enkla exempel med användning av axiomen.

Den kod, som genererats är ej optimerad. Detta och trace-utskrifterna gör att tiderna, som angivits vid exemplen, ej är relevanta och bör därför bortses ifrån.

```
RELATION
(FAR 2 ((AA) (AA)) (MANY ONE))

FAR
                                                    372 MS
************************************************
RELATION
(BROR 2 ((AA) (AA)))

BROR
                                                    253 MS
************************************************
RELATION
(SON 2 ((AA) (AA)) (ONE MANY))

SON
                                                    266 MS
************************************************
PUT
(TESTFX AXIOM ((FAR X Y) (BROR Z Y) (SON Z X)))

((FAR X Y) (BROR Z Y) (SON Z X))
                                                    114 MS
************************************************
AXCOMPILE
(TESTEX ASSERT)

TESTEX
                                                    808 MS
************************************************
PPGETP
(FAR ASSERTDEF)


  <LAMBDA (N* X Y)
    (PROG (Z ZQUE)
          (AXNAME TESTEX)
      M1  (BIND (QUOTE (Z))
                (QUOTE (BROR Z Y))
                (QUOTE B0)
                (QUOTE M2))
      B1  (CONT (QUOTE (Z))
                (QUOTE (BROR Z Y))
                (QUOTE B0)
                (QUOTE M2))
      M2  (AND (APPLY* (QUOTE AUXASSERT)
                       (LIST (QUOTE SON)
                             Z X))
               (RETURN T))
          (GO B1)
      B0  (RETURN>NIL
                                                    1108 MS
************************************************
AXCOMPILE
(TESTEX ERASE)

TESTEX
                                                    782 MS
************************************************
PPGETP
(FAR ERASEDEF)
```

```
  <LAMBDA (N* X Y)
    (PROG (Z ZQUE)
          (AXNAME TESTEX)
      M1  (BIND (QUOTE (Z))
                (QUOTE (BROR Z Y))
                (QUOTE B0)
                (QUOTE M2))
      B1  (CONT (QUOTE (Z))
                (QUOTE (BROR Z Y))
                (QUOTE B0)
                (QUOTE M2))
      M2  (AND (APPLY* (QUOTE AUXERASE)
                       (LIST (QUOTE SON)
                             Z X))
               (RETURN T))
          (GO B1)
      B0  (RETURN>NIL
                                              1061 MS
**********************************************
AXCOMPILE
(TESTEX RECSEARCH)

TESTEX
                                               786 MS
**********************************************
PPGETP
(SON RECSEARCHDEF)


  <LAMBDA (N* Z X)
    (PROG (Y YQUE)
          (AXNAME TESTEX)
      M1  (BIND (QUOTE (Y))
                (QUOTE (BROR Z Y))
                (QUOTE B0)
                (QUOTE M2))
      B1  (CONT (QUOTE (Y))
                (QUOTE (BROR Z Y))
                (QUOTE B0)
                (QUOTE M2))
      M2  (AND (APPLY* (QUOTE AUXRECSEARCH)
                       (LIST (QUOTE FAR)
                             X Y))
               (RETURN T))
          (GO B1)
      B0  (RETURN>NIL
                                              1268 MS
**********************************************
AXCOMPILE
(TESTEX (MODE CLQ CONTR 1))

TESTEX
                                              1048 MS
**********************************************
PPGETP
(SON CLQDEF)


  <LAMBDA (N* G* Z X)
    (PROG (Y YQUE)
```

```
        (AXNAME TESTEX)
    M1  (BIND (QUOTE (Y))
              (QUOTE (BROR Z Y))
              (QUOTE B0)
              (QUOTE M2))
    B1  (CONT (QUOTE (Y))
              (QUOTE (BROR Z Y))
              (QUOTE B0)
              (QUOTE M2))
    M2  (AND (SAVECL (LIST (QUOTE FAR)
                           (QUOTE CLQDEF)
                           (FUNCTIONA <LAMBDA NIL
                                        (PROG NIL
                                          M1  (RETURNTRLE
                                                (APPLY* G*))
                                              (GO B0)
                                          B0  (RETURN>
                                      (G*))
                           X Y))
             (RETURN T))
        (GO B1)
    B0  (RETURN>NIL
                                          1787 MS
**********************************************
AXCCMPILE
(TESTEX (MODE OPQ CONTR 1))

TESTEX
                                          1514 MS
**********************************************
PPGETP
(SCN OPQDEF)


  <LAMBDA (N* G* Z)
    (PROG (Y YQUE)
          (AXNAME TESTEX)
      M1  (BIND (QUOTE (Y))
                (QUOTE (BROR Z Y))
                (QUOTE B0)
                (QUOTE M2))
      B1  (CONT (QUOTE (Y))
                (QUOTE (BROR Z Y))
                (QUOTE B0)
                (QUOTE M2))
      M2  (AND (SAVEOP (QUOTE N)
                       (LIST (QUOTE REVFAR)
                             (QUOTE OPQDEF)
                             (FUNCTIONA <LAMBDA (X)
                                          (PROG NIL
                                            M1  (RETURNTRUE
                                                  (APPLY* G* X))
                                                (GO B0)
                                            B0  (RETURN>
                                        (G*))
                             Y))
               (RETURN T))
          (GO B1)
      B0  (RETURN>NIL
                                          1870 MS
**********************************************
```

```
STORE
(BROR DAVID BERTIL)

***> SYSSTORE (BROR DAVID BERTIL)
<*** SYSSTORE DAVID
OK
                                                    882 MS
**********************************************
ASSERT
(FAR ADAM BERTIL)

N* 2
***> SYSASSERT (FAR ADAM BERTIL)
  ***> SYSSTORE (FAR ADAM BERTIL)
  <*** SYSSTORE ADAM
  ENTER AXIOM TESTEX
  ***> SYSFETCH (REVBROR BERTIL ?)
  <*** SYSFETCH (DAVID)
  N* 1
  ***> SYSASSERT (SON DAVID ADAM)
    ***> SYSSTORE (SON DAVID ADAM)
    <*** SYSSTORE ADAM
  <*** SYSASSERT T
<*** SYSASSERT T
NIL
                                                   3707 MS
**********************************************
FETCH
(SON DAVID)

***> SYSFETCH (SON DAVID ?)
<*** SYSFETCH (ADAM)
(ADAM)
                                                    669 MS
**********************************************
ERASE
(FAR ADAM BERTIL)

***> SYSERASE (FAR ADAM BERTIL)
  ***> SYSREMOVE (FAR ADAM BERTIL)
  <*** SYSREMOVE T
  ENTER AXIOM TESTEX
  ***> SYSFETCH (REVBROR BERTIL ?)
  <*** SYSFETCH (DAVID)
  ***> SYSERASE (SON DAVID ADAM)
    ***> SYSREMOVE (SON DAVID ADAM)
    <*** SYSREMOVE T
  <*** SYSERASE T
<*** SYSERASE T
NIL
                                                   3381 MS
**********************************************
FETCH
(SON DAVID)

***> SYSFETCH (SON DAVID ?)
<*** SYSFETCH NIL
NIL
                                                    503 MS
**********************************************
STORE
```

```
(FAR ADAM BERTIL)

***> SYSSTORE (FAR ADAM BERTIL)
<*** SYSSTORE ADAM
OK.
                                                    643 MS
*********************************************
RECSEARCH
(SON DAVID ADAM)

N* 2
***> SYSRECSEARCH (SON DAVID ADAM)
  ***> SYSTEST (SON DAVID ADAM)
  <*** SYSTEST NIL
  ENTER AXIOM TESTEX
  ***> SYSFETCH (BROR DAVID ?)
  <*** SYSFETCH (BERTIL)
  N* 1
  ***> SYSRECSEARCH (FAR ADAM BERTIL)
    ***> SYSTEST (FAR ADAM BERTIL)
    <*** SYSTEST T
  <*** SYSRECSEARCH T
<*** SYSRECSEARCH T
T
                                                   2872 MS
*********************************************
SEARCH
(SON DAVID ADAM)

N* 2
***> SYSSEARCH (SON DAVID ADAM)
  ***> SYSTEST (SON DAVID ADAM)
  <*** SYSTEST NIL
  N* 2

  ---------------------------------
  QUE (SON CLQDEF *** DAVID ADAM)
      NXTSUBQ-N (SON DAVID ADAM)
      REMP-PROC  (LAMBDA NIL (T))

  ENTER AXIOM TESTEX
  ***> SYSFETCH (BROR DAVID ?)
  <*** SYSFETCH (BERTIL)
  ***> SYSTEST (FAR ADAM BERTIL)
  <*** SYSTEST T
<*** SYSSEARCH T
T
                                                   2336 MS
*********************************************
FIND
(SON DAVID)

N* 2
***> SYSFIND (SON DAVID ?)
  ***> SYSFETCH (SON DAVID ?)
  <*** SYSFETCH NIL
  N* 2

  ---------------------------------
  QUE (SON OPQDEF *** DAVID)
      NXTSUBQ-N (SON DAVID ?)
```

```
      REMP-PROC  (LAMBDA (X) (ANSWER X))

  ENTER AXIOM TESTEX
  ***> SYSFETCH (BROR DAVID ?)
  <*** SYSFETCH (BERTIL)
  ***> SYSFETCH (REVFAR BERTIL ?)
  <*** SYSFETCH (ADAM)
  N* 1

  ---------------------------------
  QUE (REVFAR OPQDEF *** BERTIL)
      NXTSUBQ-N (REVFAR BERTIL ?)
      REMP-PROC  (FUNAR (LAMBDA (X) (PROG NIL M1 (RETURNTRUE (APPLY*
G* X)) (GO BO) BO (RETURN))) ((G* LAMBDA (X) (ANSWER X))))

<*** SYSFIND (ADAM)
(ADAM)
                                              3571 MS
**********************************************

NOTYET  TTYIEOF
****** RESET *******
```

En test körning

```
(DEFPROP PARSE
 (LAMBDA(RP)
  (PROG (Y)
        (SETQ Y (ADVANCE))
        (COND ((GET Y (QUOTE ROP)) (SETQ Y (RUN-CODE Y)) (GO LEFT))
              ((NULL (GET Y (QUOTE OP))) (ADVANCE) (GO LEFT))
              ((IS-OPOP RP Y) (SETQ OP Y) (SETQ Y NIL) (GO LEFT))
              (T (PARSE-ERROR ARG-MISSING RP Y)))
   LEFT (COND
         ((GET OP (QUOTE LOP))
          (COND
           ((GREATERP RP (GET OP (QUOTE LP))) (RETURN Y))
           (T (SETQ Y (RUN-LCODE OP)) (GO LEFT))))
         ((IS-AA Y OP) (SETQ Y (CASEAA Y OP)) (GO LEFT))
         (T (PARSE-ERROR OP-MISSING Y OP)))))
EXPR)

NIL
```

värdet av (PARSE 2) på inputen nedan

```
(DEFPROP PARSE
 (LAMBDA(RP)
  (PROG (Y)
        (SETQ Y (ADVANCE))
        (COND ((GET Y (QUOTE ROP)) (SETQ Y (RUN-RCODE Y)) (GO LEFT))
              ((NULL (GET Y (QUOTE OP))) (ADVANCE) (GO LEFT))
              ((IS-OPOP RP Y) (SETQ OP Y) (SETQ Y NIL) (GO LEFT))
              (T (PARSE-ERROR ARG-MISSING RP Y)))
   LEFT (COND
         ((GET OP (QUOTE LOP))
          (COND
           ((GREATERP RP (GET OP (QUOTE LP))) (RETURN Y))
           (T (SETQ Y (RUN-LCODE OP)) (GO LEFT))))
         ((IS-AA Y OP) (SETQ Y (CASEAA Y OP)) (GO LEFT))
         (T (PARSE-ERROR OP-MISSING Y OP)))))
EXPR)

NIL
```

kontroll mot def av parse

```
0100   PROCEDURE PARSE ( RP ) ;
0200           BEGIN
0300                   LOCAL Y ;
0400                   Y := ADVANCE ( ) ;
0500                   IF GET ( Y , ' ROP ) THEN
0600                           Y := RUN-CODE ( Y ) & GOTO LEFT
0700                   ELSE IF NULL ( GET ( Y , ' OP ) ) THEN
0800                           ADVANCE ( ) & GOTO LEFT
0850                   ELSE IF IS-OPOP ( RP , Y ) THEN
0875                           OP := Y & Y := NIL & GOTO LEFT
0900                   ELSE PARSE-ERROR ( ARG-MISSING , RP , Y ) ;
1000           LEFT :
1100                   IF GET ( OP , ' LOP ) THEN
1200                           IF RP > GET ( OP , ' LP ) THEN RETURN Y
1300                           ELSE Y := RUN-LCODE ( OP ) & GOTO LEFT
1350                   ELSE IF IS-AA ( Y , OP ) THEN
1500                           Y := CASEAA ( Y , OP ) & GOTO LEFT
1600                   ELSE PARSE-ERROR ( OP-MISSING , Y , OP )
1700           END ;
```

input till parse (för def av parse nedan)

```
@100:10000
00100   (DE PARSE(RP)(PROG(Y)
00200     (SETQ Y (ADVANCE))
00300     (COND((GET Y (QUOTE ROP))(SETQ Y (RUN-RCODE Y)) (GO LEFT))
00400          ((NULL(GET Y (QUOTE OP)))(ADVANCE)(GO LEFT))
00500          ((IS-OPOP  RP Y)(SETQ OP Y)(SETQ Y NIL)(GO LEFT))
00600          (T(PARSE-ERROR ARG-MISSING RP Y)))
00700     LEFT
00800       (COND((GET OP (QUOTE LOP))
00900             (COND((GREATERP RP (GET OP OLP))(RETURN Y))
01000             (T(SETQ Y (RUN-LCODE OP)) (GO LEFT))))
01100            ((IS-AA Y OP)(SETQ Y (CASEAA Y OP)) (GO LEFT))
01200            (T(PARSE-ERROR OP-MISSING Y OP$
01300   (DE ADVANCE()(SETQ OP (READ-TOKEN$
01400   (DE IS-AA(01 02)(EQUAL 02 OLPAR$
01500   (DE IS-OPOP(X Y)(EQUAL Y ORPAR$


01600   (DE CASEAA(X ARG)(PROG(RES)(SETQ RES (PARSEAA 100))
01700     (RETURN(COND(RES(CONS X (REMNARY RES)))(T(LIST  X$
01800   (DE PARSEAA(RP)(PROG(Y)
01900     (SETQ Y OP)
02000     (COND((GET Y OROP)(SETQ Y (RUN-RCODE Y))(GO LEFT))
02100           (T(ADVANCE)(GO LEFT)))
02200   LEFT
02300     (COND((GET OP OLOP)
02400           (COND((GREATERP RP (GET OP OLP))(RETURN Y))
02500                 (T(SETQ Y (RUN-LCODE OP))(GO LEFT))))
02600           ((IS-AA Y OP)(SETQ Y (CASEAA Y OP)) (GO LEFT))
02700           (T(ERROR OP-MISSING Y OP$
02800   (DE RUN-RCODE(X)(APPLY(GET X OROP) NIL (ALIST$
02900   (DE RUN-LCODE(X)(APPLY(GET X OLOP) NIL (ALIST$
03000   (DF ALIST(L ASS)ASS)
03100   (DF ROP(L ASS)(PROG2(PUTPROP(CAR L) T OOP)
03200     (CAR L)
03300     (PUTPROP(CAR L)  (APPEND O(LAMBDA NIL)(CDR L))OROP$

03400   (DF LOP(L ASS)(PROG2 (PUTPROP(CAR L) T OOP)

03500     (CAR L)
03600     (PUTPROP(CAR L)  (CADR L)OLP)
03700     (PUTPROP(CAR L)  (APPEND O(LAMBDA NIL)(CDDR L))OLOP$
03800   (ROP BEGIN (MKPROG (REMNARY(PARSE O$
03900   (ROP LOCAL (REPCOM OLOCAL (PARSE 2$
04000   (ROP PROCEDURE (MKPROC(PARSE 2)(PARSE 2$
04100   (ROP LPAR (PROG (RES)(SETQ RES (PARSE 4))(ADVANCE)(RETURN RES$
04200   (ROP RETURN(LIST ORETURN (PARSE 4$
04300   (ROP ' (PROG(RES)(SETQ RES (ADVANCE))(ADVANCE)(RETURN
04400          (LIST OQUOTE RES$
04500   (ROP GOTO(LIST OGO (PARSE 4$
04600   (ROP IF (MKCOND(PARSE 4)
04700                  (COND((EQUAL OP OTHEN )(PARSE 4))(T(PARSE-ERROR IF OP T
))))
04800                  (COND((EQUAL OP OELSE)(PARSE 4))(T NIL$
04900   (LOP END -1 NIL)
05000   (LOP ; 1 (COND((EQUAL (CAR Y)O;)(NCONC Y (LIST(PARSE 2))))
05100                (T(LIST O; Y (PARSE 2$
05200   (LOP RPAR 3 NIL)
```

```
05300   (LOP THEN 3 NIL$
05400   (LOP ELSE 3 NIL)
05500   (LOP & 5 (PROG(L R )(SETQ L Y)(SETQ R(PARSE 4))
05600     (RETURN(COND((EQUAL(CAR R)O&)(CONS O&(CONS L(CDR R))))
05700       (T(LIST O& L R$
05800   (LOP : 5 (LIST O: Y (PARSE 2$
05900   (LOP := 5 (LIST OSETQ Y (PARSE o$
06000   (LOP COMMA 5 (COND((EQUAL (CAR Y) OCOMMA)(NCONC Y (LIST(PARSE 6))))
06100                        (T(LIST OCOMMA Y (PARSE 6$
06200   (LOP > 15 (LIST OGREATERP Y (PARSE 16$
06300   (DE MKPROG(L)(PROG(RES)
06400     (COND((EQUAL (CAAR L) OLOCAL) (RPLACA L (CDAR L)))
06500     (SETQ RES (CONS OPROG L))
06600   LOOP
06700     (COND((NULL L)(ADVANCE)(RETURN RES))
06800           ((EQUAL (CAAR L )O:)
06900                    (RPLACD(CDDAR L)(CDR L)) (RPLACD L(CDDAR L))(RPLACA L
DAR L))))
07000     (SETQ L (CDR L))
07100     (GO LOOP$

07200   (DE MKCOND(A B C)
07300     (APPEND(LIST OCOND (CONS A (REMNARY B)))
07400             (COND((EQ (CAR C) OCOND)(CDR C))(T(LIST(CONS T(REMNARY C$
07500   (DE REPCOM(TYPE X)
07600     (COND((EQ (CAR X) OCOMMA)(RPLACA X TYPE))
07700           (X (LIST TYPE X))     (T(LIST TYPE$
07800   (DE MKPROC(HEAD TAIL)(PUTPROP(CAR HEAD) (LIST OLAMBDA (CDR HEAD) TAIL
APR$
07900   (DE REMNARY(X)(COND((MEMBER(CAR X)O(COMMA ; &))(CDR X))(T(LIST X$
08000   (DE READ-TOKEN()(READ$
```

Kommentarer till den genererade koden. I axiomet är literalen (BROR Z Y) ett sk sidovillkor. Varje sådant sidovillkor bildar ett bind-cont par, där bind utför access till databasen, i detta fall genom att bygga upp en kö zque för axiomen framåt och yque för axiomen bakåt och cont står för "continuation" och kommer att ta ett elementet från kön och gå vidare i axiomet osv. Access till databasen sker normalt genom endast en explicit sökning (sysfetch eller systest) eller med rekursiv sökning (sysfind eller sysrecsearch).I exemplet ställs alltså frågan (REVBROR y ?) vid framåtsökningen och (BROR x ?) vid bakåtsökningen. För erase, assert och recsearch sker fortsättningen sedan rekursivt, motsvarande aux-funktion anropas, som utför någon operation (auxerase raderar bort slutsatsen ur databasen) och fortsätter sedan med nya axiom om sådana finnes.

Vid search och find kommer en kö av subfrågor att behöva administreras. Vid denna användningen av axiomet är det ej klart vilken av literalerna, som skall bilda subfrågorna. Användaren kan därför med hjälp av CONTR-parametern ange denna literal. Övriga literaler blir sidovillkor. I exemplen kommer första literalen att generera subfrågan. För att klara av detta har införts en sk. remainder-procedur. Idéen beskrivs utförligt i ref. 3. Till varje subfråga associeras en remainder-procedur, och denna appliceras på varje svar, som subfrågan genererar.I exemplet finns en trace, som skriver ut kön, nästa subfråga och dess remainder-procedur. I exemplet med search finns på kön en subfråga (SON CLQDEF :::: DAVID ADAM), vilket skall tolkas, som att subfrågan är "finns SON(DAVID, ADAM) lagrat", CLQDEF anger att det är en sluten fråga (clq = closed question) :::: står för dess remainderprocedur.

Ett mer illustrativt exempel med remainder-proceduren är

((Q X Y)(R X Z)(S Z Y)(T Y X))

Antag vi använder axiomer bakåt för öppna frågor och att vi vill att andra literalen (R X Z) skall generera subfrågor. (T y ?) medför först att sidovillkoret (S Z Y) kommer att generera en kö av alla z den känner till. För varje sådant z, ställer den subfrågan (R ? z). Denna subfråga läggs på kön, och dess remainder-funktion kommer att innehålla kod för

att testa för varje svar x, som subfrågan finner, om (Q x y) är lagrad. Remainder-proceduren kommer alltså att innehålla ett bind-cont par (vilket kommer att reduceras ned till ett rent anrop till systest, som kollar om (Q x y) är lagrad. Remainder-proceduren kommer att bli ett funarg-uttryck, där bland annat y blir en fri variabel, som binds vid subfrågans generering. Evalueringen av remainder-proceduren sker ju först då subfrågan funnit ett x, och då den alltså appliceras på detta x.

Vi har alltså lyckats att samla ihop x-en i (R X Z) på ett icke-deterministiskt sätt. Finns ett x fortsättes i axiomet (remainder-proceduren evalueras), misslyckas vi eller om vi vill ha fram alla x, fortsätter vi sökningen efter ett nytt x etc.

Remainder-proceduren kan även initialiseras, så att man kan söka efter endast första förekomsten av ett svar. ("existerar det något x, som ..... och i så fall ge mig detta") eller söka efter de fem första förekomsterna eller efter alla svar. Även extra filtrering av svaren kan utföras med detta.

Konverteringen av PCDB (ca 375 LISP-funktioner) från LISP F1 till INTERLISP gick ganska smärtfritt. De allra flesta ändringarna kunde utföras automatiskt med hjälp av den macro-expandern, MAKRO, som beskrivs i avsnitt 8c.

Fortsatt arbete med PCDB. Under 1974 skall C-versionen "låsas". Smärre arbete återstår dock och då främst att "trimma" in kompilatorerna att även klara av alla möjliga fall där H-uttryck ingår.

En riktig text på systemets effektivitet får man ej förrän all genererad kod först optimeras av redfun eller liknande programmanipulerande program och sedan kompileras till maskinkod av LISP-kompilatorn. Denna kompilator finns nu under 1974 tillgänglig på INTERLISP, så experiment och analys kommer att utföras. Fler experiment behöver också utföras för att utröna användbarheten av ett generellt slutsatsdragningsprogram, som PCDB. Vissa jämförelser med andra existerande AI-språk (se avsnitt 16) kommer också att utföras.

Referenser

(1) Anders Haraldson
PCDB - A Program which Generates Procedures for Maintaining a Data Base of Formulas in Predicate Calculus,
DLU, nov 1973.

(2) Lennart Drugge
Documentation of the PCDB package, alphabetic list of functions,
DLU, mars 1972.

(3) Eric Sandewall
Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs.
Proc. of third IJCAI, Stanford Univ., Stanford, Calif. (1973).

9b GIP/GUP
~~Generell~~ in-och utmatning av propertylistor

De flestaav LISP:s grundenheter, atomerna, har en egenskapslista, där programmeraren kan lagra flera olika informationsmängder som är associerade till atomen. Varje sådan informationsmängd kallas en egenskap. För att man ska kunna specifiera en bestämd egenskap på en egenskapslista är varje egenskap lagrad tillsammans med en speciell atom, kallad indikator, som alltså motsvarar ett termnamn i ett record. Den väsentligaste skillnaden är att nya indikatorer kan tilläggas dynamiskt under körningen, och att man alltså inte behöver deklarera från början vilka indikator/egenskapspar man vill ha.
Exempel:

> Atomen ADAM kan under indikatorn SÖNER ha egenskapen (KAIN ABEL). Skrivs förenklat ADAM : SÖNER = (KAIN ABEL). Detta är ett bekvämt sätt att lagra faktum att Adam har sönerna Kain och Abel.

INTERLISP-systemet har faciliteter för att utföra enkla operationer på egenskapslistor, t ex lägga upp egenskap under viss indikator, ändra egenskaper och söka efter viss egenskap. Om man ska mata in eller skriva ut stora mängder egenskapslisteinformation måste man dock använda ganska klumpiga och arbetskrävande metoder. GIP/GUP-programmen är avsedda att råda bot på detta och erbjuda LISP-användare möjlighet att mata in och ut stora mängder egenskapslisteinformation på ett någorlunda bekvämt och överskådligt sätt.

Vid inmatning anropas funktionen GIP med bl a ett argument som talar om hur indata ska vara organiserad, vilka kontroller av indata GIP ska utföra samt hur lagring av information ska ske. Därefter läser GIP information, som användaren förväntas ge på den form, som specificerats i argumentet.

Exempel på indata:

```
ADAM : SÖNER = KAIN, ABEL /  skulle kunna åstadkomma samma
struktur som i föregående exempel.
/ anger att indata är slut.
```

Man kan också samtidigt ge egenskaperna egenskaper i sin tur.

Exempel:

```
ADAM : SÖNER = KAIN : FAR = ADAM |,
                      ABEL : FAR = ADAM /
```

I detta fall åstadkommer vi alltså:

```
ADAM:SÖNER=(KAIN ABEL)
KAIN:FAR=ADAM
ABEL:FAR=ADAM
```

| anger att inmatningen till KAIN:s egenskapslista är klar.

Vid inmatningen anropas funktionen GUP med ett argument, som talar om vilka atomers egenskapslistor vi är intresserade av, samt ett argument som anger hur informationen är lagrad, vad som ska skrivas ut och hur utskriften ska organiseras. GUP skriver därefter ut den önskade informationen.

Exempel:

Det som matades in i föregående exempel skulle kunna skrivas ut sålunda:

```
ADAM:
=====
SÖNER=
    KAIN
    =====
      FAR= ADAM;,
    ABEL:
    =====
      FAR= ADAM;
```

De argument som GIP och GUP anropas med kan innehålla en stor mängd styrinformation, vilket gör programmen relativt generella. T ex behöver man inte nödvändigtvis använda egenskapslistor för lagringen, utan användaren kan själv inom vissa gränser välja sin egen lagringsform.

Den stora mängden möjlig styrinformation gör dock programmen i deras parameterstyrda form långsamma. En stor del av exekveringstiden går åt till att testa på variabler som i allmänhet har defaultvärden. GIP/GUP har därför tagits som testexempel för den allmänna principen för "redskapslådan" vid DLU att sikta på programgeneratorer hellre än generella, parameterdrivna program. Experimenten med detta redovisas i sektion 8a.

Arbetet på själva GIP/GUP-programmen är avslutat och dokumenterat i DLU 73/10.

9c. TOP-parsern

Ett återkommande problem i smådatabastillämpningar är vilken inmatningsnotation man skall använda för fakta till databasen, och för förfrågningar. Speciellt för det senare är det ofta praktiskt att välja en notation av "algol-typ", där man alltså får ha dels prefix- och infixoperatorer, dels nyckelordssekvenser av typ algol:s för ... := ... step ... until ... do . Såväl i exemplet med SNV-databasen (se avsnitt 12d) som exemplet DLUORG (avsnitt 12 c) har detta behov uppkommit. Vidare och som ytterligare ett specialfall finns samma behov när man vill ge en "sockrad" programnotation för LISP eller motsvarande, ss. i PLAST-systemet (se avsnitt 4f) och i REC (avsnitt 4 g).

Härigenom uppkommer alltså behovet av en generell parser, som kan analysera sådana uttryck enligt givna specifikationer. Tidigare har ett program IKP utvecklats och använts vid DLU för detta ändamål (jfr föregående årsrapport). Under arbetet på REC (se avsnitt 4g) har Torgny Tholerus utvecklat en annan parser. Denna har sedermera av Mats Nordström överförts till LISP-systemet och döpts till TOP (för Torgnys Operator-Parser), samt använts för flera tillämpningar. Den huvudsakliga skillnaden är att IKP styrs av parameterar, som ligger

som liststrukturer, medan TOP delvis styrs av hjälp-procedurer som användaren skriver. I avvägningen mellan de två programmen har TOP bedömts som lättare att använda. Effektiviteten har jämförts i interpreterad kod i INTERLISP, varvid TOP var effektivare, men å andra sidan är IKP skriven med stora PROG-uttryck som direkt avser att kunna kompileras effektivt, men som blir mycket missgynnade vid interpretering. Jämförelsen är alltså ej slutförd.

En beskrivning av principerna för TOP gavs i avsnitt 4g, som behandlade språket REC. Utom för REC har TOP använts som inöversättare för PLAST (avsnitt 4f), i SIMLISP-projektet (kapitel 6), och för formelmanipulationssystemet REDUCE (utfört vid Mats Nordströms vistelse i Salt Lake City). En mycket närbesläktad parser har oberoende utvecklats vid MIT och använts för MACSYMA-projektet (formelmanipulation) och som översättare från ytspråk till LISP.

Vid användningen av TOP har denna visat sig vara utmärkt för experimentändamål, då man enkelt vill skaffa sig en parser till även relativt komplicerade språk. Så har t ex en fullständig parser för algol -60 definierats inom ISMLISP-projektet. Däremot är den i sig okänslig för omgivningen till en sats eller ett uttryck, varför dess diagnosmöjligheter är relativt svaga. Exempelvis skulle en sats av typ A + goto D accepteras vid konventionell användning av TOP. I många fall kan detta dock botas genom att lägga in syntaxkontroll i konstruktionsfunktionerna.

9d. TABLEOUT

I avsnitt 9b behandlades problemet med in- och utmatning av egenskapslistor, och där beskrevs de program (GIP/GUP etc) som framställts för att lösa detta. De utnyttjar en indenterings (= radindragnings-) uppställning, som medger presentation även av rätt komplicerade strukturer. I vissa fall har man en enklare data-struktur, och önskar en kolumnuppställning med en egenskap i varje kolumn. För detta fall har ett ytterligare program, TABLEOUT, framställts. Programmet är till för att skriva ut egenskapslistor som består av en lista av indikator/

egenskaper. Komplikationer på listorna, bestående av att egenskaperna har egna egenskapslistor, tillåtes ej. Listorna skrivs ut i tabellform, med indikatorerna i tabellhuvudet och egenskaperna i tabellen, vilket gör dem relativt lättlästa. Då programmet är enkelt och erbjuder få valmöjligheter är det ganska snabbt.

10. Programredskap och metoder på naturligt-språk-nivå

Programredskap under denna rubrik förväntas kunna användas dels i renodlade forskningsprojekt på naturligt-språk-"förståelse", (se kapitel 14), dels som hjälpmedel för olika tillämpningsprojekt. Bl a avses nätverksparsern (avsnitt 10c) användas i SNVDATA-projektet (avsnitt 12 c).

10a. PCF-2 (=predikatkalkylformulering av semantiska strukturer.)

PCF-2 är ett föreslaget "dataspråk" för att representera "naturligt-språk-information" i databasen. Notationen har utvecklats allteftersom. En systematisk och jänförelsevis uttömmande notation utkom i januari 1972. Denna har under året modifierats på ett antal punkter. Mer arbete har emellertid lagts ned på att bygga ut implementeringen av PCF-2, kallad SEPAC, och att göra experiment med denna. Se avsnitt 10b, 10d och 14c nedan. Slutligen har ett meta-system (databas och program) som beskriver PCF-2-notationen, och kan producera förteckningar av olika typ, tjäna som underlag för programgenerering i PCDB (se avsnitt 9a), etc., skrivits.

10b. SEPAC (= program för hantering av enkla meningar) och NLTOP (= topploop för interforce med SEPAC)

De centralaste delarna av PCF-2, för hantering av enkla egenskaper ("Peter är stor") och händelser ("Peter ser huset") har en struktur som skulle bli alltför ineffektiv om den sköttes enbart med PCDB-genererad kod. Därför har under tidigare år ett programpaket SEPAC skapats som utför lagring, hämtning och slutsatsdragning enligt de centrala delarna av PCF-2, men som på viktiga punkter innehåller handkodade LISP-funktioner. (Den organisation som PCDB förutsätter har dock bibehållits, så att koppling till PCDB-kompilerade versioner av andra delar av PCF-2 skall vara lätt att utföra).

Programmet SEPAC har kraftigt utvidgats under året, och har kommit så långt att vissa experiment har kunnat köras. Dessa har avsett dels

enkelt frågebesvarande, vilket är den triviala tillämpningen, dels disambiguering, där vissa uttryck såsom vissa flertydiga uttryck upplöses; se avsnitt 14b.

För det fortsatta arbetet planeras:

- Ytterligare ett antal mindre experiment, för att verifiera att programmet kan användas för olika aspekter av naturligt-språk-"förståelse", och för att få en uppfattning om vad som saknas.
- Implementering av ett system för tillskottseditering (jfr DLU 1972, avsnitt 3c) på hög nivå, så att olika uppsättningar av programtillägg och programändringar, gjorda av olika personer, lätt kan mergas.
- Fortsatt utbyggnad av systemet med olika delar av PCF-2-notationen, och med ytterligare slutsatsdragningsregler.

10c. Nätverksparser o -grammatik

Woods nätverksparser, samt den kompilator och editor som skrivits vid DLU finns tidigare beskrivna i DLU 1972 och DLU 1973. Under året har visst underhållsarbete bedrivits i samband med att systemet lagts över på INTERLISP, men mesta arbetet har legat på att skriva en grammatik som är kompatibel med SEPAC-paketet. (jfr avsnitt 14c)

10d. FORIN (= formelmeningar)

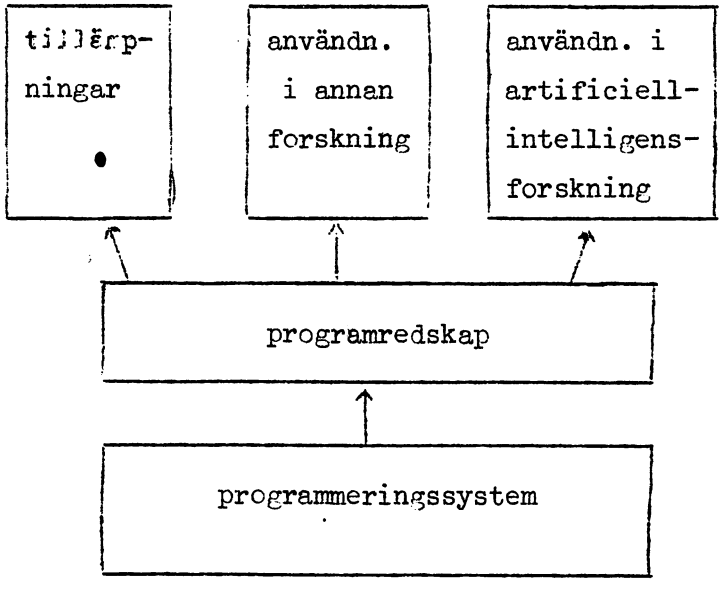
I föregående årsrapport omnämndes ett program NL, som var avsett att användas vid experiment med SEPAC-programmet och med enkel textstrukturanalys. Programmet medgav inmatning av en anspråkslös, något engelskliknande formelnotation. Under 1973 har detta program vidareutvecklats och sönderfallit på två: en topploop, NLTOP, som sköter den löpande textkontakten med användaren (inläsning av meningar, etc.), och ett program FORIN som analyserar de nu något snyggare formelmeningarna, och ger dem till SEPAC. Programmet har visat sig mycket användbart inom sitt avsedda användningsområde, och har bl a använts för de ovan nämnda SEPAC-experimenten med frågebesvarande och disambiguering.

ANVÄNDNINGAR AV SMÅDATABASER

11. Bakgrund och översikt

Datalogilaboratoriets verksamhet har sin rot i arbete på frågebesvarande system, heuristisk sökning, mm som bedrevs vid institutionen (utan särskild forskningsgrupp) under tiden 1965-70. Detta var alltså arbete som faller inom forskningsområdet artificiell intelligens. När DLU bildades 1970 ansåg vi att metodiken för frågebesvarande system borde kunna generaliseras och användas även på i tiden mer näraliggande tillämpningar, dels inom annan forskning och dels för praktiska problem, utanför universitetet. Vi valde då beteckningen små databaser för det forsknings-område vi ville täcka in.

Redan från början såg vi alltså arbetet uppdelat på dels smådatabasmetodik, dels på tre typer av användningar: i tillämpningar (utanför universitetet), för artificiell-intelligens-forskning, och användningar för annan forskning. Metodikarbetet sönderföll snart i arbetet på programmeringssystem och på programredskap. Strukturen i arbetet illustreras i följande figur:



Vi har försökt realisera dessa typer av användningar parallellt med varandra. Bland tillämpningar dominerar f n ett försök med frågesystem för naturvårdsbas, som görs i samarbete med IBM och med forskare vid Naturvårdsverket; se avsnitt 12c. Efterföljare till detta projekt planeras för närvarande. Bland användningar i annan forskning dominerar ATMAN-projektet, som beskrivits ingående i tidigare årsrapporter, och som kort refereras i kapitel 13. Användningar i artificiell-intelligens-forskning avser i första hand arbete på naturligt-språk-"förstående" system, motsvarande den engelska termen "computer understanding". Häri ingår frågebesvarande system som en del. Årets arbete på detta område refereras i kapitel 14 och (vad gäller programredskap) även i kapitel 10.

I kapitel 12 nedan har även medtagits några experiment med programredskap, som gjorts enbart för att testa dessa, och på exempel uppfunna inom DLU.

11a. Översikt över möjliga tillämpningar

Under vårterminen 1973 sammanställdes vid Datalogilaboratoriet såsom diskussionsunderlag ett PM om möjliga, jämförelsevis näraliggande tillämpningar av smådatabaser. Detta PM, som alltså var att betrakta som en samling förslag från vår sida, utsändes till ett antal personer verksamma i näringsliv och förvaltning med förfrågan om synpunkter på vilka förslag som syntes orealistiska eller ointressanta, och vilka som kunde tillkomma. De inkomna svaren var ganska få, men i gengäld mycket innehållsrika. Svaren har inarbetats i den ursprungliga texten, och resultatet presenteras nedan.

Ett antal mer långsiktiga typer av tillämpningar föreslogs i kapitel 1 av denna årsrapport, och i tidigare årsrapporter.

Förslagen till mer näraliggande tillämpningar delade vi upp i tre huvudklasser:

A. Fristående smådatabaser, dvs situationer där smådatabaser och tillhörande program användes oberoende av annan databehandling.

B. Programmeringshjälpmedel, där smådatabaser användes för att underlätta framställning eller användning av datorprogram.

C. Databashjälpmedel, där smådatabaser användes för att understödja programmering för och användning av stora (= konventionella) databaser och databanker.

De olika klasserna presenteras var för sig. Vi lägger stor vikt vid punkt (C), och diskuterar f n försökstillämpningar som skulle falla under denna punkt.

A. Fristående smådatabaser

Följande någorlunda generella tillämpningsklasser har diskuterats:

A1. Konstruktionsarbete (Computer-aided design) för digitallogik

Administration av databas av tillgängliga komponenter och kretselement; analys av kretsförslag som ges av användaren; en konstruktion av kretsar till givna specifikationer; kostnadsberäkning mm; renritning av schema.

A2. "Slit- och släng"-program, företrädesvis av interaktiv typ, där en liten databas, manipulation av formler eller strukturer är väsentliga.

Här avses sådana tillämpningar, där man idag föredrar att sköta datamaterialet manuellt i form av skrivna listor o dyl, men där man gärna skulle lägga över materialet i maskinen när körkostnaderna sjunker och maskinen blir interaktivt tillgänglig eller eljest lättare åtkomlig. Det förutsätts att programmen är av måttlig storlek, och inte behöver kopplas till andra program eller datamängder, och att programmen alltså kan skrivas "rätt av" utan någon omständlig planering.

Exempel från en arbetsplats (kontorsmiljö):

- Register över egna lokaler; placering av personal, inventarier, telefoner, osv.
- Nycklar: vem har vilken nyckel; vilken nyckel passar var?

- Datoriserad almanacka, som kommer ihåg användarens bokade tider och de uppgifter som han skall göra före viss dag, och som påminner honom vid lämpliga tillfällen.

Exempel från en datacentral el dyl:

- Kundregister
- Register över magnetbandarkivet
- Översikt över aktuella program, vem som är ansvarig, etc.

Detta är endast ett fåtal exempel, flera enkätsvar understryker att listan kunde göras mycket lång.

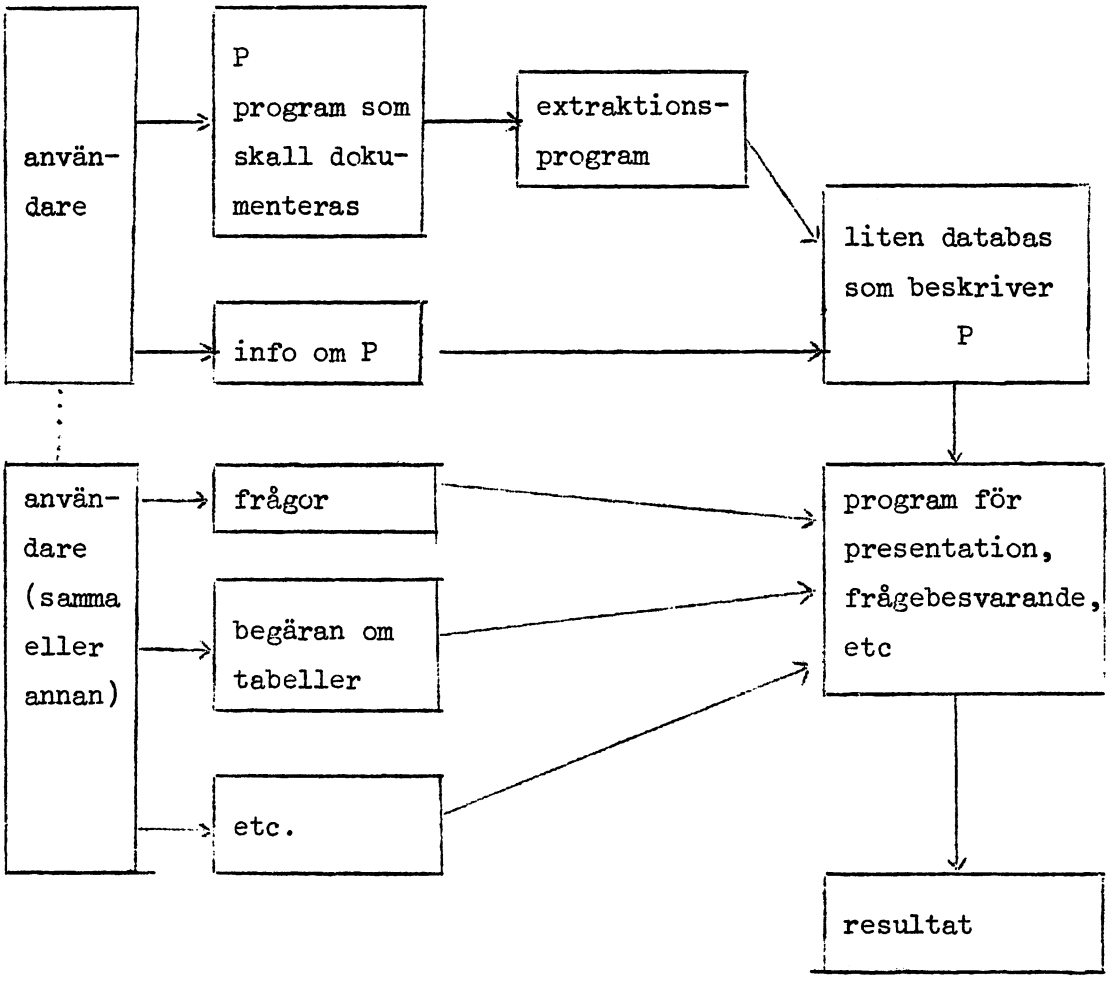
B. Programmeringshjälpmedel

B1. Programdokumentation. Dokumentationsarbete är en betungande och ofta eftersatt del av programmeringsarbetet. En liten databas som lagrar information om ett program eller programsystem kunde användas för att:

- generera tabeller över t ex anropsstruktur mellan rutiner, referenstabell (var sätts resp används viss variabel, var sker hopp till visst läge, etc)
- generera välstrukturerade flödescheman där triviala detaljer utelämnas
- skapa tabeller över dataflödet mellan olika variabler (vilka indatavariabler, och ev vilka mellanresultat, kan påverka värdet av viss variabel)
- interaktivt svara på frågor av samma typ som i ovanstående tabeller (så att inte hela tabellen behöver skrivas ut)

Programdokumenterande program av denna typ skulle huvudsakligen vara användbara som konstruktionshjälpmedel när programmet skrivs och när det senare skall ändras.

För att utföra sådana uppgifter måste den lilla databasen innehålla rätt mycket information från det (de) givna programmen. Denna bör erhållas genom att programmet självt matas in till ett program som extraherar den väsentliga informationen, samt genom att användaren av dokumentations-programmet kompletterar med ytterligare upplysningar som inte direkt kan utläsas av koden i det program som skall dokumenteras.



Figur till B1: Dataflöde

B2. Dokumentation av programbibliotek. Många standardprogram (ex rutiner för plottning, rutiner för graphics i Fortran) används genom att man anropar dem med ett stort antal parametrar, som vanligen ges som konstanter i varje anrop. En liten databas skulle kunna hålla reda på:

- betydelsen hos varje parameter
- tillåtna värden för varje parameter; tillåtna kombinationer av värden
- (ev) hur parametrar skall väljas för att önskat resultat skall uppnås (av intresse om parametrarnas betydelse är programorienterad snarare än problemorienterad)

samt användas för att

- interaktivt bistå en användare av sagda program
- kontrollera att anropen till rutinerna från ett givet användarprogram är meningsfulla.

Vidare skulle den lilla databasen kunna veta vilka program som finns i programbiblioteket, deras ursprung, begränsningar, etc, vilka program som utför liknande uppgifter (användbart för frågor ss "det här programmet gör inte riktigt vad jag vill ha gjort, vilket kan jag använda istället?"), resurskrav, etc. Detta kunde i första hand användas för att interaktivt besvara frågor från användare som vill finna och använda ett program för en viss uppgift.

Denna föreslagna tillämpning skiljer sig från den föregående (B1) framförallt genom att det här är fråga om ett användarhjälpmedel (användare av program i programbiblioteket) snarare än av ett konstruktionshjälpmedel. Men naturligtvis finns det även gränsfall, där användaren samtidigt vill modifiera biblioteksprogrammet, och bör därför försöka få ett integrerat system för uppgifterna B1 och B2.

B3. Interface-system. Sedan man tagit fram ett programsystem för en viss tillämpning, måste man med dagens teknik vanligen ha en programmerare eller motsvarande som preparerar körningar, specifierar styrparametrar, etc. I många situationer (simuleringar, databanker) vill man emellertid helst att experter på resp problemområde skall kunna använda programmet direkt. Detta förutsätter att indata till programmet kan anges på ett enkelt sätt, utan kännedom om programmets inre funktionssätt, och med användning av begrepp ur resp tillämpningsområde. Detta ställer stora krav på den programdel ("användar-interfacet") som läser in specifikationer (styrparametrar eller motsvarande) och tolkar dem. Bland annat måste interfacet då innehålla viss faktakunskap om resp problemområde. Åtminstone i svårare fall förefaller det lämpligt att isolera interface-programmet, och utforma det med smådatabasmetodik.

Exempel B3a. Det nu pågående projektet "Frågesystem för naturvårdsdatabas" som beskrivs i avsnitt 12c.

Exempel B3b. För lösning av partiella differentialekvationer under besvärliga omständigheter (oregelbunden rand mm) finns ett antal kända metoder. För varje givet problem är det därför en rätt rutinmässig uppgift att skriva ett program för den uppgiften. Å andra sidan är variationen i metoderna så stor att man knappast kan göra ett enda generellt program. Man borde emellertid kunna göra ett bibliotek av programfragment, och sedan i varje given tillämpning välja relevanta fragment och redigera ihop dem. Man kan förstås klara sig med ett konventionellt editeringsprogram för detta, men trevligare skulle vara att ha ett interface-program, som läser in en programorienterad specifikation av uppgiften (i matematiska termer) och utfärdar rätt kommandon till editeringsprogrammet.

Denna tillämpning har föreslagits av Jan Johansson, avd för numerisk analys, Uppsala, och Anders Beckman, DLU. Även andra tillämpningar av samma metodik kan vara möjliga.

B4. Programgeneratorer. I vissa ofta återkommande tillämpningar, t ex sortering, har man sedan länge föredragit att använda programgeneratorer framför generella program. Det är i princip klart att en programgenerator möjliggör större effektivitet än det generella programmet, men det är samtidigt mer besvärligt att skriva en sådan.

När man skriver en programgenerator, består en väsentlig del av uppgiften i att manipulera formeluttryck: ett program är ju uppbyggt av formler. En annan väsentlig del är att hålla reda på deklarationer, lägen, etc i det hittills genererade programmet, samt att hålla reda på och använda givna specifikationer, som ju kan vara ganska utförliga. Slutligen måste en programgenerator ofta innehålla kunskap om resp tillämpningsområde. Allt utom möjligen formelmanipulationen är här väsentligen smådatabasproblem. Ett programmeringssystem som är inriktat mot smådatabaser och formelmanipulation, av typ LISP, bör därför göra det betydligt lättare att skriva det genererande programmet, än

om detta skall skrivas i konventionella språk. Däremot torde det ofta vara motiverat att låta det genererade programmet vara i ett konventionellt språk.

En närmare diskussion av olika tillämpningar av programgenererande och programmanipulerande program återfinns i kapitel 19 (sid 146 ff) av Datalogilaboratoriets förra årsrapport "DLU 1973".

C. Hjälpmedel vid programmering för och användning av stora databaser

Arbete med stora databaser inkluderar programmering, databasvård, etc. Samtliga de tillämpningar av smådatabaser som programmeringshjälpmedel, vilka omnämndes ovan under B, är även tillämpliga när programmeringen avser en stor databas. Det ovan nämnda exemplet B3a avser interface med ett program för att hantera en medelstor databas.

I det följande föreslås ytterligare några typer av tillämpningar, som mer specifikt inriktar sig på arbete med stora databaser.

C1. Databasdokumentation. En dokumentation av en stor databas bör innehålla:

- uppgift på vilken information som finns i den stora databasen, uttryckt i problemorienterade termer (t ex med hjälp av den logik som utvecklats för lagring av naturligt-språk-information).
- uppgift på vilka datamängder som finns i den stora databasen, uttryckt i datatekniska termer (filer, dataset etc)
- uppgift på vilken (vilka) datamängder som uttrycker respektive information, och hur den uttryckes.

Uppgifterna på lagrad information måste kompletteras med annan kunskap om problemområdet för att vara användbara. Uppgifterna på förekommande datamängder bör innehålla alla normala datororienterade attribut: storlek, struktur, tillkomstdatum, föräldra-datamängder, lagringsmedium, etc. Uppgift på kopplingen bör i första hand ange hur viss information kan lagras och hämtas bland data (anges ev som en procedur), men det kan också vara intressant att åt andra hållet ange vilken information som innehålles i vissa data.

Dokumentationen av datamängderna är användbar för följande ändamål:

- för databasadministratören, som vill ha god översikt över vad som finns i databasen,
- för programmeraren, som skall skriva program som anknyter till denna databas,
- för program som analyserar förväntade åtkomsttider, kostnader och andra aspekter av prestanda för den givna (föreslagna) databasorganisationen.

I de båda förra fallen kan informationen göras tillgänglig för användaren genom tabeller eller annan liknande presentation, eller genom interaktiva förfrågningar när viss information behövs.

Dokumentation av informations-innehållet och hur det är lagrat, är av intresse för de två första av ovanstående ändamål, samt även för:

- hjälpinformation för interface-system enligt B3 ovan, som översätter problemorienterade frågor till motsvarande programkommandon,
- underlag för analys av vilken information som finns lagrad, men ej är effektivt tillgänglig i den givna databasorganisationen.

C2. Databashistorik. Som en utvidgning av föregående punkt C1 föreslås en liten databas som även innehåller datamängdernas historik, dvs vilka förändringar som gjorts under den senaste tidsperioden. Ur denna historikdatabas skall användare och tillämpningsprogram kunna få uppgifter såsom "har operation A(vilken skall utföras i slutet av varje månad) gjorts för denna månad?". Om många programmerare arbetar på samma databas är det ett självklart intresse att kunna få snabba och säkra svar på sådana frågor.

Ytterligare en tillämpning av databashistoriken är för recovery vid t ex maskinfel: man vill ju alltid ha så mycket redundans i en databank att man kan återskapa förlorade delar efter ett haveri, men metoden för att återskapa dessa kan få bero av i vilket läge och på vilket sätt haveriet inträffade. Ett program som automatiskt väljer rätt metod för att återskapa förlorade delar måste känna till förekommande datamängder och deras inbördes relationer. Relationerna uttryckes ofta bäst genom att ange datamängdernas informationsinnehåll. Informationen enligt C1 ovan behövs alltså. Men recovery-programmet måste också känna till databasens senaste historia.

C3. Informationsdistribution. Betrakta en situation där ett flöde av data (t ex mätvärden eller statistiska uppgifter som insamlas regelbundet) dels skall ackumuleras i en databas, dels distribueras till ett antal avnämare. Man har många olika avnämare i en stor organisation; olika avnämare skall ha olikt utförliga data, med olika täthet mellan rapporteringstillfällena; vissa avnämare vill ha systematiska rapporter; andra vill bara ha varningar när vissa alarmvillkor är uppfyllda; vissa avnämare vill även ha återblick över tidigare data av samma typ ur databasen etc.

I en sådan situation består systemeringsarbetet i att utreda hur inkommande data är strukturerade, vilken information de innehåller, vilka användare och användarbehov som föreligger, och hur de effektivast skall tillfredsställas. Den vanliga arbetsmetoden är att först göra systemeringen, och sedan använda resultatet därifrån som underlag för programmeringen.

I vissa fall är emellertid situationen så föränderlig att denna metodik inte kan användas. Nya avnämare kan anmäla sig, och gamla avnämare kan ändra sina uppgivna informationsbehov. Då kan systemeringen t o m vara inaktuell innan programmeringen är färdig.

Ett alternativ är då att i en liten databas ("distributionsdatabas") lagra dels en beskrivning av strukturen hos inkommande data (jfr tillämpning C1 ovan), dels en beskrivning av kända avnämare och deras informationsbehov. Dessa uppgifter lagras naturligtvis på ett generellt sätt, så att ytterligare avnämare efter behov lätt kan införas. Därefter användas dessa data till att styra informationsdistributionen. Detta kan möjligen ske genom att man har ett generellt program, som styrs av distributionsdatabasen, men bättre är att ha en programgenerator, som utifrån distributionsdatabasen genererar ett lämpligt program för informationsdistribution. Denna generering kan då göras om varje gång avnämarstrukturen ändrats.

Detta arbetssätt kan karakteriseras som fortgående systemering, i den

bemärkelsen att systemeringsresultatet = distributionsdatabasen fortlöpande ändras. Eftersom systemeringsresultatet här måste uttryckas helt strikt, ställer detta arbetssätt kanske större krav på systemeraren än vid konventionellt arbetssätt, men i gengäld får man ett system som lättare kan anpassas till nya krav.

Det som ovan under B4 sades om programgeneratorer är tydligen direkt tillämpbart på denna tillämpning.

Så som tillämpningen ovan skisserats, har man bara en informationskälla, men många informationsmottagare. Problemet kan naturligtvis generaliseras till det fall då man har flera, ev interagerande informationsgivare, men uppgiften att utforma den beskrivande databasen och generera lämpligt program blir då väsentligt svårare. Det förefaller lämpligt att t v bara diskutera det fall då man har ett inkommande informationsflöde till systemet. (Dock kan man förstås tillåta att det finns flera informationskällor, om informationen därifrån integreras, "mergas", av ett separat, handkodat program).

C4. Pilotsystem. När ett databassystem utformas genom systemering (antingen på konventionellt sätt, eller med fortlöpande systemering enligt ovan) framställer man en abstrakt specifikation av problemet, som sedan användes som underlag för programmeringsarbetet. Ibland kan denna metod kompletteras med, eller t o m ersättas av arbete med ett pilotsystem. Idén är då att skriva ett program i ett bekvämt programmeringssystem (gärna ett som använder virtuell minnesteknik och är interaktivt tillgängligt) som utför samma sak som det slutgiltiga systemet skall göra, men som bara dimensioneras för en liten datamängd. Detta pilotsystem användes för att experimentera med möjliga datastrukturer, kontrollera att alla frågor som skall kunna besvaras, även effektivt kan besvaras, etc. När pilotsystemet är uttestat, skriver man det definitiva systemet på basis av erfarenheterna därifrån.

Det programmeringsspråk och programmeringssystem som skall användas för ett sådant pilotsystem måste tillfredsställa två motstridiga krav: å ena sidan måste det vara så mycket bekvämare än det slutgiltiga programmeringssystem, att det ger en väsentlig förbättring jämfört med

att koda det slutgiltiga systemet direkt. Å andra sidan bör det vara så likt det slutgiltiga systemet, speciellt vad gäller datastrukturerna, att erfarenheterna från pilotsystemet är tillämpliga vid arbetet med det definitiva systemet.

Vi gör i princip bedömningen att ett smådatabassystem av LISP-typ (inkl PLAST) bör vara lämpligt för sådana pilotsystem, men vill gärna verifiera detta genom praktiska försök.

Om metoden med pilotsystem visar sig användbar, kan man tänka sig vidarutvecklingar, t ex att utifrån pilotsystemets program plus ytterligare specifikationer (t ex av datakvantiteter, frågetäthet) försöka generera produktionsprogrammet automatiskt.

12. Tillämpning och experiment

12a. CICERON

CICERON, som beskrevs utförligt i DLU -73, är ett LISP-program som understödjer uppläggning och utnyttjande av en databas innehållande information om gatunät, byggnader m m i en stad. Jämfört med tidigare dokumenterade version har under året endast mindre modifieringar skett. Sålunda har konversationsloopen utvecklats något och programmet tar nu som inmatning satser i "fritt" format avslutade med punkt, frågetecken eller utropstecken. Ex:

VI BEFINNER OSS VID CENTRALEN.
HUR KOMMER MAN TILL NÄRMASTE HOTELL?

Vidare har programmet under året konverterats till LISP 1.6, vilket medfört möjligheter till interaktiv körning på DEC 10 vid Stockholms datacentral och tillåtit att ett flertal användare provat CICERON.

Härvid har följande svagheter i analysen av inmatade satser visat sig:

- Programmet saknar med vissa undantag för närvarande möjligheter att identifiera objekt som av användaren stavas eller benämns på ett sätt som avviker från den interna formen. Problem av detta slag har förekommit oftare än förutsett.
- Nära förknippat med föregående är behovet att ibland kunna upptäcka att ett okänt ord förmodligen är exempelvis ett gatunamn.
- Inmatade satser analyseras isolerat och referenser av typen DEN eller DIT kan inte upplösas.
- Vissa förstagångsanvändare avviker från den valda konversationsmodellen (ung: samtal mellan två personer) genom att använda exempelvis kommando- eller programmeringsspråksliknande formuleringar.
- Avsaknaden av ett formellt frågespråk utgör en ofelbar inspiration för användare att försöka testa vad systemet egentligen klarar. Programmet har dock begränsade möjligheter att ge en meningsfull

respons på inmatade satser där användaren går utanför systemets ram.

Sammanfattningsvis kan sägas att "naturligt språk"-delen i CICERON har visat sig vara kanske alltför enkel, men att väsentliga förbättringar borde kunna göras med måttligt programmeringsarbete och ökning av informationsmängden. För övrigt har programmet fungerat tillfredställande inom ramen för den valda ambitionsnivån.

12b. DLUORG: Ett tillämpningsexempel på PLAST

För att undersöka PLAST:s kapacitet (tekniskt, inte effektivitetsmässigt) och för att göra en antydan om dess förutsättningar och möjligheter kodades under våren 1973 ett halvstort tillämpningsexempel i PLAST. Syftet med programmet var att hantera en smådatabas, vilket ju är avsikten med PLAST som programmeringsspråk, och på grund av bristande utrymmesresurser i den aktuella implementationen snarare än brist på datamaterial eller manipulationskraft tenderade databasen att bli en liten smådatabas. Detta hindrade dock inte att de avsedda aspekterna på PLAST belystes, och experimentet gav som helhet ett tillfredsställande resultat. En dokumentation av programmet återfinns i*; här följer endast en kortfattad redogörelse för dess målsättning och struktur tillsammans med några exempel:

Målsättning.

Databasens uppgift var att upprätthålla en struktur över DLU:s interna organisation vad det gäller forskningsverksamheten. Den skall då kunna ge svar på frågor av typen "sök alla projekt som person NN deltagit i", "sök de deluppgifter inom projekt PP som pågått under tiden TT", "när stödde sponsor SS projektet PP" eller "hur är personen NN relaterad till anslaget AA".

Databasens organisation.

Databasen är en nätverksliknande struktur mellan noder på olika hierarkiska nivåer, där nivåerna anger nodernas klass, som kan vara t ex sponsor, projekt eller person, och där noderna då är objekt av dessa klasser. Objekten tillordnas en mängd egenskaper, som kan ange t ex

---

* DLU 73/11.

deras eventuella tidsutsträckning samt naturligtvis deras relationer till andra objekt. Relationerna representeras som listor av objekten i den närmast över- resp. underordnade klassen. För tidsrepresentationen har den grova indelning månad (per år) använts, och ett tidsintervall utgörs då av en lista av begynnelse och slutmånad.

Programmet.

En grundläggande procedur läser in data från hålkort, som stansats enligt en given mall, och genererar databasen utifrån dessa. Övriga huvudprocedurer ägnar sig åt utsökningar av större eller mindre komplexitet, eller utökar databasen med nya objekt. En funktion SÖK(kl, o) finner t ex alla objekt av klassen kl som är relaterade till objektet o, medan SÖK(kl, o, t) dessutom lägger det villkoret på de utsökta objekten, att dessa skall ha pågått under någon del av tiden t. Proceduren NÄR(o1, o2) ger tiden för den givna relationens upprätthållande. Med hjälp av funktionen RELATION(o1, o2) fås den kedja av relationer som leder från o1 till o2. Varje utsökning (eller fråga) ger ett resultat i list-form, men användaren kan specificera att en tabellarisk uppställning också genereras. Systemets generella sökfunktioner kan naturligtvis även sammanställas till rutiner för mer komplicerade sökningar enligt användarens önskan.

För att i vissa fall erhålla en aptitligare notation, har PLAST-syntaxen utökats med operatorer för att ange tidsintervall, en möjlighet till syntaxmodifikation som ingår i språket. Hur denna notation ser ut framgår av exemplen.

Exempel:

Okomplicerade fakta finns som egenskaper hos objekten:

```
        PLASTEX:"DEFINITION ;
   <<KONSTRUKTION AV ETT STÖRRE TILLÄMPNINGS EXEMPEL>>
Proceduren SÖK.
        SÖK(PERSONER,SNVDATA) ;
   <HÄGGLUND HARALDSSON>
        SÖK(ANSLAG,CICERON) ;
   <FOA4>
```

SÖK-ning med tidsvillkor och med tabellutskrift.

```
        SÖK(PROJEKT,NIL,UNDER VT 1973) ;

   PROJEKT CICERON
   FRÅN JANUARI 1973 TILL FEBRUARI 1973

   PROJEKT PLAST
   FRÅN JANUARI 1973 TILL JUNI 1973

   PROJEKT SIMULALISP X
   FRÅN JANUARI 1973 TILL JUNI 1973

   ...

        SÖK(DELUPPGIFTER,PLAST,1973 TILL MAJ 1973) ;

   DELUPPGIFT PLASTMOD2
   UNDER JANUARI 1973

   DELUPPGIFT PLASTDEF
   FRÅN FEBRUARI 1973 TILL MARS 1973

   ELUPPGIFT PLASTEX
   FRÅN APRIL 1973 TILL MAJ 1973
```

proceduren NÄR.

```
        NÄR(WILLEN) ;

   <WILLEN ÄR ETT PERMANENT OBJEKT>

        NÄR(INTERFORM,STU) ;

   SPONSOR STU

   FRÅN JULI 1970 TILL DECEMBER 1971
```

Proceduren RELATION.

```
        RELATION(IBM,HÄGGLUND) ;

   SPONSOR IBM
        ANSLAG IBM4
             PROJEKT SNVDATA
                  DELUPPGIFT SNVANALYS
                       PERSON HÄGGLUND
```

```
DELUPPGIFT SNVREDSKAP
   PERSON HÄGGLUND
DELUPPGIFT SNVKOD1
   PERSON HÄGGLUND
```

Ett nytt objekt tillförs på följande sätt.

```
NEWOBJEKT(IMPLSPRÅK,KLASS ÄR PROJEKT, ANSLAG ÄR STU4,
        TID ÄR UNDER VT 1973,DEFINITION ÄR
        <<STUDIUM OCH UTVÄRDERING AV IMPLSPRÅK>>,
        ANSVARIG ÄR "BECKMAN) ;
<CBJEKT IMPLSPRÅK AV KLASS PROJEKT RELATERAT TILL STU4 ÄR DEFINIERAT>
```

En enkel procedur för att beskriva ett objekt kan definieras som följer.

```
%FUNCTION OBJBESKR(X);
LOCAL I;
PRINT NEWLINE,"BESKRIVNING, "AV,X::KLASS,X;
FOR I IN X::KLASS::INDS DO;
PRINT NEWLINE,I,":,X:I;
NEXT I;
PRINT(NEWLINE) & RETURN X;
%END;
```

Den kan användas som t ex

```
OBJBESKR(STU);
```

```
BESKRIVNING AV SPONSOR STU
KLASS : SPONSOR
ANSLAG :<STU1 STU2 STU3 STU4>
NAMN :<STYRELSEN FÖR TEKNISK UTVECKLING>

STU
```

Slutligen definieras ett antal ytterligare operatorer, så att frågor kan ställas på ett språk, som ytligt sett liknar naturlig engelska. Frågorna kan då se ut på t ex följande sätt.

```
SEARCH PROJEKT;
SEARCH DELUPPGIFTER FROM JANUARI 1973 UNTIL MARS 1973;
WHEN WAS PLASTDOK;
SEARCH DELUPPGIFT RELATED TO PLAST DURING HT 1972;
```

WHEN WAS INTERFORM RELATED TO WILLEN;
HOW WAS INTERFORM RELATED TO WILLEN;

Slutsatser

Projektet har resulterat i ett demonstrationsexempel som är tillräckligt stort för att vara av något intresse, och samtidigt tillräckligt begränsat för att lätt kunna överblickas.

Särskilt bör observeras att det framtagna programmet i första hand användes som ett slutet system, dvs. att användaren ger data och ställer frågor till detta i en förutsedd notation, men att användaren också kan utvidga notationen på icke-trivialt sätt, och att han kan skriva till egna småprocedurer som anropar det givna systemet. För sådana småprocedurer, som skall läggas till av användaren, är det särskilt viktigt med ett lättlärt och enkelt språk, och vi tror att PLAST då passar bra.

12c. Frågesystem för ekologisk databas.

Under 1973 yppade sig möjligheten för Datalogilaboratoratoriet att få tillgång till erfarenheter och data från ett samarbetsprojekt mellan IBM Svenska AB och Naturvårdsverket (SNV). Detta utgjorde en lovande utgångspunkt för ett tillämpningsprojekt vid DLU där metoder och program för små-databasbearbetning kunde testas i en verklig problemomgivning.

IBM-SNV-projektet innebar bl a att IBM utvecklade ett experimentellt relationsdatabas-system IS/1-0 som användes av biologer vid SNV för att göra komplexa analyser av primärdata rörande främst fiskmigration. Den akumulerade databasen bestod huvudsakligen av tidsserier av mätdata insamlade endera direkt av SNV eller införskaffade från SMHI. Inom projektets ram utfördes och dokumenterades ett 75-tal databasbearbetningar mestadels innebärande statistisk bearbetning eller presentation av data i form av tabeller, plottar o d.

Det av IBM utvecklade IS/1-0 systemet tillåter användaren att se sina

datamängder som relationer, och han ska därmed kunna operera på data utan att behöva veta hur filerna är fysiskt strukturerade. Nya funktioner kan man enkelt tillföra språket genom att programmera en modul i PL/I och definiera lokal syntax för funktionen. All bearbetning i IS/1 sker via en stack som kan manipuleras med olika kommandon. Icke-trivial programmering i IS/1:s frågespråk krävs normalt åtminstone för frågeställningar som berör flera relationer.

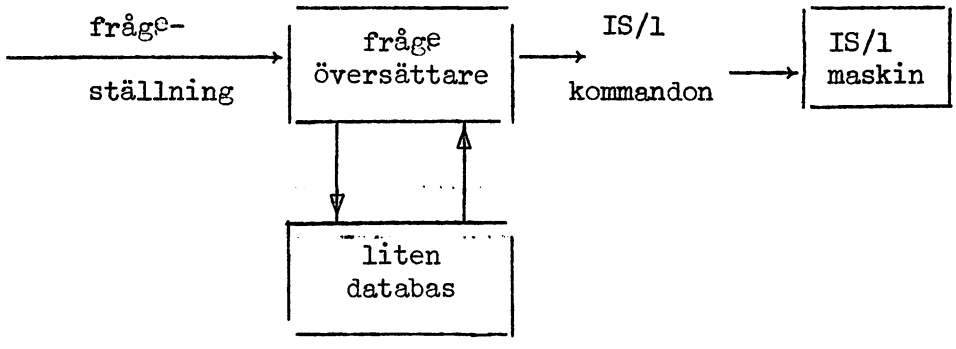
Ett naturligt önskemål i många databastillämpningar är att förkorta "avståndet" mellan användaren och den lagrade informationen. Detta är ett område där vi tror att små-databas-tekniken kan bidra till att lösa en del problem. I IBM-SNV-projektet har vi en väldefinierad mängd användareformulerade frågeställningar och motsvarande IS/1-program. Att försöka eliminera behovet av en programmerare och automatisera överföringen av en fråga till program var en i våra ögon intressant uppgift.

Våren 73 startades vid DLU ett projekt, stött av IBM, med i huvudsak denna inriktning. Den explicita målsättningen är att förenkla kommunikationen mellan en ämnesområdesexpert och en databas genom att utveckla en frågespråks-interpretator som med hjälp av bl a meta-information om databasen genererar databas-manipulerande program från resultat-orienterade frågor. Vår angreppsmetod för att studera detta problem är att i LISP implementera ett program som klarar en stor klass av frågorna i IBM-SNV-projektet med ett icke-proceduralt frågespråk som så nära som möjligt anslutar sig till de SNV-dokumenterade formuleringarna. Vårt program ska generera sekvenser av IS/1-kommandon som enligt överenskommelse med IBM kommer att kunna testköras i det tidigare använda IS/1-systemet. Det förtjänar dock att här påpekas att vi i detta sammanhang inte på något sätt vill ta ställning till hur IS/1:s frågespråk borde se ut. Från vår synpunkt är detta system en svart låda, en "databasmaskin" för vilken vi kompilerar program.

Som ett viktigt hjälpmedel under utvecklingsskedet har inom ramen för vårt projekt skrivits ett LISP-program som simulerar IS/1-systemet. Kombinerat med en liten testdatabas medger detta att genererad kod direkt kan exekveras och därmed en logikkontroll fås. En intressant modifika-

tion av denna teknik skulle vara att i produktionssystemet inkludera möjligheter till testexekvering av frågor varvid möjligen orimligt tidskrävande eller oinformativa körningar skulle kunna undvikas.

I vårt användar-interface kan två logiska komponenter urskiljas, dels själva frågeöversättaren, dels den lilla databas som innehåller information om den stora databasen, allmänna fakta om tillämpningsområdet och annan kunskap som kan utnyttjas av frågesystemet.



I ovanstående framställning har begrepp som interpretering, kompilering och programgenerering använts omväxlande och det kan kanske vara på sin plats att lite närmare utreda graden av interaktivitet i vårt system. En användare avses alltså inte kunna direkt bearbeta den stora databasen, utan interaktiviteten inskränker sig till att han kan konversera med frågesystemet under utvecklandet av sin frågeställning. Denna modell har valts av projekttekniska skäl, trots att den skarpa åtskillnaden syns oss en smula onaturlig och bl a leder till viss dubbellagring av information.

Frågespråksdelen. Uttag av enkla fakta ur en databas erbjuder normalt inga större svårigheter, utan problemen uppstår först när man vill göra komplicerade sammanställningar och bearbetningar av sina data. Sådana frågeställningar kan i allmänhet beskrivas som kombinationer av datamängdsoperationer och applicerande av funktionsprogram av typen plottning, medelvärdesberäkning etc. Tillgången till sådana funktionsfaciliteter betraktas som given, varför vår huvudsakliga uppgift blir att administrera skapandet av nya datakonstellationer genom:

(a) uppdelning av datamängder
(b) kombination av datamängder
(c) direkt tillförande av nya data

under hänsynstagande till selektionsvillkor och med möjlighet till transformationer av ingående enkla data (ex. aritmetiska operationer).

En normal fråga till databasen kan anses bestå en två delar:

- Specifikation av vilken operation man vill utföra.
- Specifikation av parametrar till denna operation.

Om operationen exempelvis är skapande av tabell, kan parametrarna vara vilka data som ska ingå i tabellen, eventuella transformationer av dessa data, villkor för att data ska tas med etc. Den interna representationen av en fråga är i form av operation plus parameterspecifikationer. Det är vår ambition att testa olika typer av frågespråk, med varierande grad av språklig frihet, men hela tiden översätta till den angivna notationen. Från denna representation sker sedan generering av databasmanipulerande program, i vårt fall IS/1 kommandon.

Lilla databasen. Denna bör i princip medge lagring och utnyttjande av all slags information som kan användas för att underlätta frågeställarens arbete. Som exempel på sådant som kan ingå kan nämnas:

- Index över i stora databasen ingående datamängder (relationer och domäner).
- Klartext- eller annan högnivå-beskrivning av lagrade data.
- Kvantitativ och kvalitativ information om lagrade data.
- Strukturell information om hur olika datamängder är relaterade till varandra, exempelvis härledda data, alternativa åtkomstmöjligheter till samma data etc.
- Allmän information om problemområdet som kan användas för att ge en meningsfull respons på inexakta frågor.
- Möjlighet för användare att definiera egna begrepp.
- Möjlighet att spara frågor för senare referens till dessa med modifiering av någon parameter.

I den tidigare nämnda SNV-databasen består grundmaterialet av ett begränsat antal tidsserier av mätdata där relativt mycket information samlats in vid varje tidpunkt. Från detta material har sedan successivt nya datakonstellationer (relationer) bildats och bearbetats. Detta betyder att samma data kan logiskt återfinnas på ett antal olika ställen i databasen. En av uppgifterna för den lilla databasen blir att styra användaren till att utnyttja den för honom lämpligaste relationen. Dock bör påpekas att denna funktion i ett produktionssystem måste samordnas med IS/1 systemet eftersom detta själv administrerar härledda relationer och inte alltid lägger ut dessa explicit. En annan punkt där erfarenheterna från IBM-SNV-projektet tyder på ett behov är möjligheten att upprepa en tidigare fråga i något modifierad form. För ett givet databasinnehåll visar det sig nämligen att en stor del av frågorna kan inordnas i några få grundmönster.

Arbetet vid DLU med att utveckla ett användar-interface för en ekologisk IS/1-databas har hittills haft tyngdpunkten på problemet att generera databasmanipulerande program från en resultat-orienterad fråga. Ett jämförelsevis formellt nyckelordsbaserat frågespråk har provats. Från detta språk har parsing och översättning till intern representation skett med hjälp av den parser IKP, som finns som programredskap vid DLU och tidigare använts i språket PLAST. (Jfr föregående årsrapport).

Ett exempel ska få illustrera något av hur vårt program för närvarande fungerar. Antag att vi i en databas som innehåller provfiskedata och hydrologiska data har en relation CATCH som är definierad över bl a domänerna SPECIES = fiskart, DATE= datum och NUMBER = fångst av viss art viss dag samt relationen WATERTEMP med bl a domänerna DATE = datum och SURFTEMP, BOTTOMTEMP = temperatur på olika djup viss dag. En användare som är intresserad av hur fiskfångsten beror av vattentemperaturen kanske vill sammanställa antalet fångade abborrar med medelvattentemperaturen för varje dag och kan då skriva exempelvis:

```
CREATE PEARCHTEMP FROM CATCH, WATERTEMP WITH DATE, NUMBER,
TEMP = (SURFTEMP + BOTTOMTEMP)/2 FOR EACH DATE WHILE SPECIES =
ABBORRE;
```

Man kan alltså ange vilka relationer man vill utgå ifrån, vilka domäner som ska ingå i en nyskapad relation, hur värden i dessa ska bildas förutsatt att vissa villkor är uppfyllda osv. Från denna fråga genereras sedan en sekvens av IS/1 kommandon som selekterar, transformerar och kombinerar data ur berörda relationer.

Den använda IKP-parsen är endast avsedd för ett rätt skikt och formellt språk, och är därför svårt att modifiera för tillämpning på annat än rena formeluttryck och nyckelord-argument sekvenser. Därför avses nu experiment göras med nätverksgrammatik och nätverksparser för att kunna tillåta ett flexiblare frågespråk. Parallellt med detta kommer lilla databasen att utvecklas vidare. För närvarande innehåller den huvudsakligen information om i databasen lagrade relationer och domäner. Dessutom finns där administration av ställda frågor som bl a tillåter att man upprepar en tidigare fråga något modifierad.

12d. HOUSE - Testexempel på PCDB

HOUSE är ett trebetygsarbete utfört av Ingvar Kvarnbäck. Målsättningen var att skapa ett större testexempel och testa B-versionen av PCDB (beskrivet i kap. 9).

HOUSE beskriver ett hus. Första delen utgör en statisk beskrivning av huset, såsom rummens placering gentemot varandra, dörrar som förbinder rummen etc. I andra delen införs en dynamisk beskrivning med hjälp av situationsbegreppet. Ytterligare information införs då för att beskriva vad som händer då en person förflyttar sig i huset, dvs då nya situationer uppstår.

Exempel på relationer, funktioner och axiom:

INOM	rum x hus	beskriver att ett rum finns i huset
ÄR	rum x prop	beskriver att ett rum har en viss egenskap, exempelvis lägesförhållande till andra rum.
läge-till	rum x loc →	prop funktion för att generera ett lägesförhållande.

Man alltså kan t ex lagra

ÄR(VARDAGSRUMMET, läge-till(HALLEN,ÖSTER))

dvs "vardagsrummet ligger till öster om hallen."
Omvändningen att "hallen ligger till väster om vardagsrummet" kan erhållas genom ett axiom

ÄR(x,läge-till(y,l))∧MOTSATT(l,ll)⊃ÄR(y,läge-till(x,ll))

För att beskriva att en dörr är öppen eller stängd i en situation finns en relation

LÄGE situation x loc x dörr

Om glasdörren är stängd vid situation s_0 lagras

LÄGE(SO,STÄNGD,GLASDÖRREN)

En funktion

ändra situation x loc x dörr ⟶ situation,
skapar en ny situation i vilken dörrens läge ändrats.

Ett axiom för att beskriva detta är

LÄGE(s,l,dr)∧MOTSATT(l,ll)⊃LÄGE(ändra(s,l,dr),ll,dr)

Rapporten (DLU73/16)innehåller alla deklarerade relationer (12 st) och funktioner (5 st) samt alla axiom (20 st). Bifogat finns även all den LISP-kod, som PCDB har genererat.

Testexemplen visade vissa felaktigheter och brister i denna PCDB-version och erfarenheten från dessa tester har även legat till grund för den omarbetade C-versionen.

13. Användningar i annan forskning

13a. ATMAN

Atmanprojektet har de senaste fem åren arbetat med att utveckla ett system för simulering av kausala modeller av socialt handlande. Under 1973 har en databasmonitor färdigprogrammerats kring vilken vi nu bygger databasen. I maj 1974 avser vi publicera fem rapporter om detta projekt, bland annat även beskrivningen av monitorn.

Planering, konversation och naturligt-språk-kommunikation är ännu försummade sidor av simuleringssystemet. Vi skall under 1974 framför allt rikta in oss på att förbättra de simulerade beslutsenheternas förmåga att planera sitt handlande.

14. Naturligt-språk-arbete

Naturligt-språk-analys är relaterat till smådatabasmetodik dels såsom ett framtida hjälpmedel (naturligt språk skulle i många fall vara den idealiska metoden att kommunicera med ett smådatabassystem), dels såsom en omedelbar tillämpning (i experiment med naturligt-språk-analys idag behöver man ha tillgång till små databaser i vilka man lagrar dels grammatik, dels en liten ordlista, dels den information som överförs i eller kan utläsas från den inlästa texten).

Arbete på analys och "förståelse" av naturligt språk medelst dator utförs inom forskningsområdet artificiell intelligens, och under de senaste åren har en livlig utveckling ägt rum inom denna forskning, som bl a fått impulser från den nya generationen av "ultraspråk" (jfr kapitel 16).

Det arbete som bedrivs vid DLU siktar främst på att söka isolera och behandla väsentliga delproblem. I detta avsnitt redogöres för två sådana arbeten, nämligen på en konversationsmodell och på disambiguering. Det i förra årsrapporten föreslagna projektet (för läsåret 1973/74) på en "nyfiken databas" har också påbörjats.

14a. Konversationsmodell

Vid studium av datoranalys av naturligt språk ("computer understanding") har man vanligen begränsat sig till att behandla en sats eller mening i tagit. Det finns emellertid också en struktur på högre nivå, den som bl a kallas "disposition" (av en skriven text). Även denna är av stor betydelse för att datorn skall kunna "förstå" texten.

I detta arbete har vi sökt utforma och implementera en enkel modell för samtal. Arbetet har grundats på uppfattningen att ett samtal kan betraktas som sammansatt av olika samtalsfragment, som interagerar med varandra. Mönstret för dessa samtalsfragment finns lagrade hos de olika deltagarna, men varje deltagare har sin specifika uppsättning av mönster.

Ett sådant mönster för ett samtalsfragment kallas en jigg. En jigg kan betraktas som ett litet program som interpreteras av en deltagare, och där det som sades i föregående tidssteg är inmatning till programmet. Så formulerad kan denna modell för samtal tänkas täcka såväl enklare konversation (av typ "kafferast" eller "danstillställning") som mer komplicerade dialoger (t ex debatter). För den konkreta implementeringen begränsade vi oss dock till mycket enkel och stereotyp konversation, där man kan anta att mönstren för hur man talar är få till antalet och enkla till strukturen.

I konversationen kan ett begränsat antal deltagare förekomma. Var och en av dessa har sina specifika egenskaper och reagerar olika på vad som sägs i diskussionen. Medlen tillåter att man kopplar in sig själv som deltagare i konversationen.

Utförande

Simulerigen är tidsstyrd. I varje tidssteg kan det bara förekomma en Communication Act (CA) dvs att någon säger något till någon, samt ev en eller flera Observed Acts (OA), dvs en yttre händelse. Begränsningen att bara en CA kan inträffa i varje tidssteg innebär alltså att deltagarna inte kan prata i munnen på varandra. Varje deltagare har en uppsättning jiggar. Dessa jiggar har en dubbel funktion. De används dels får att "producera" en CA, dels för att se om den andre parten i samtalet följer vad som förväntas av honom.

En jigg är i det enklaste fallet ett antal repliker där varannan skall sägas av A och varannan av B. Ser nu jiggarna hos A och B likadana ut kommer detta samtalsfragment att fungera väl: A säger något till B, B hittar en jigg som börjar med vad A sade, och ger sin replik, A tittar efter i sin jigg och ser att B sade vad som väntats och kan fortsätta osv. Problem kan naturligtvis uppstå om A och B inte har samma uppsättning av jiggar, så att B inte adekvat kan svara på A:s utsaga. Detta är ju något som också inträffar i naturliga livet om folks referensramar skiljer sig åt för mycket.

I jiggen kan också finnas andra instruktioner som gör att replikerna inte kommer i tur och ordning. Det finns t ex jiginstruktioner av typ GO TO, samt villkorskonstruktioner. Antag att en jigg innehåller:

(*) if x är glad then x säger "HURRA" till y

Låt sedan x vara bundet till ADAM och y till EVA. När ADAM skall interpretera (*) tittar han efter i sin databas för att se om han är glad, och om han är det säger han "HURRA" till EVA. Eva i sin tur interpreterar (*) genom att "lyssna" efter om ADAM säger "HURRA", och om han gör det drar hon slutsatsen att ADAM är glad och lagrar detta i sin databas.

En OA kan slumpmässigt genereras av systemet t ex att en tavla ramlar ner, att telefonen ringer och dessa händelser kan då naturligtvis avbryta de samtal som just håller på.

Nuvarande status

Systemet har skrivits för PL-360 lispen och finns f n inte på INTERLISP. Programmet har fungerat för mindre testexempel men något försök att köra det med ett större antal jiggar och mera komplicerade "karaktärer" hos deltagarna har ännu inte gjorts.

14b. Disambiguering

Dessa experiment kördes med programmen SEPAC och FORIN (jfr avsnitt 10b och 10d). Programmen fick "meningar" i vilka ingick uttryck ss.

John´s hand
John´s wallet
The father of John

samt

A glass bottle
A whisky bottle
A horse shoe

etc. Databasen innehöll viss plausibilitetsinformation (ss. "en kroppsdel må vara del av en människa"). Programmet skulle välja en

föredragen tolkning av resp. uttryck ("den hand som är en del av John", "den plånbok som John äger", etc.).

14c. Programsammankoppling

Under hösten hölls vid DLU en seminarieserie om olika aspekter på naturligt-språk-arbete. Inom ramen för den har ett antal program-system diskuterats: sådana som skrivits på DLU, sådana som nu finns tillgängliga här och sådana som skulle kunna tas hit. Speciellt har möjligheterna att koppla ihop olika program till ett större system diskuterats. De program det gäller är bl a nätverksgrammatiken, SEPAC-paketet, Simmons' utöversättarprogram och konversationsövervakaren (simulatorn). Eftersom programmen funnits i diverse olika LISP-dialekter och under olika stadier av utveckling, har själva hopkopplingen hittills stannat vid att SEPAC-paketet nu fungerar tillsammans med nätverksgrammatiken. Simmons' program är under uppläggning på INTERLISP. Det är dock avsett för generering av meningar på engelska, varför en del arbete återstår innan det passar in i ett större system.

15. Diverse

Till detta kapitel har samlats ett antal trebetygsuppgifter vilka utförts under året vid Datalogilaboratoriet, men som ej passar in i kapiteluppdelningen för denna rapport.

15a. Experiment med PPS = Planning Problem Solver

PPS är en metod för heuristisk sökning som definierar hur en meritfunktion för sökningen kan konstrueras om man känner sannolikheterna för vissa tillståndsövergångar. I föreliggande arbete har PPS-metoden provats på en mycket enkel tillämpning, nämligen 15-spelet, och med en jämförelsevis liten och enkel rymd av tillstånd, vilket gör det möjligt att beräkna hela meritfunktionen (för alla tillstånd) innan sökningen påbörjas. För de körda exemplen inom denna tillämpning tycktes PPS-metoden dock ge ingen eller föga förbättring jämfört med tidigare använd metod.

15b. Helsidesformat i FORTRAN

Detta är ett trebetygsarbete utfört av Kai Sippel. Vid utskrift av tabelliknande information kan det i de konventionella programmeringsspråken vara svårt och besvärligt att beskriva hur man vill att en utskriftssida skall se ut. I FORTRAN:s FORMAT-satser finns det exempelvis normalt ingen möjlighet att använda ett variabelt antal fält, och det normala är att man genererar en rad, skriver ut denna, genererar nästa rad etc.

Detta program underlättar planeringen av en helsida. Användaren deklarerar och namnger först ett antal fält, talar om var dessa fält börjar och slutar på raden, samt på vilka rader de kan förekomma. Dessutom ges information vilken typ av data (integer, floating, character etc) som fältet kan bestå utav. Vissa fält kan innehålla fix information, exempelvis för rubriker. Dessa fält kommer alltid att skrivas ut vid sidbyte.

Användaren har sedan till sitt förfogande två subrutiner för att göra utmatningen i FORTRAN. Till den ena anger man vad som skall skrivas ut, på vilket fält samt eventuellt på vilken rad, och detta medför att rutinen sköter om all eventuell konvertering efter de tidigare givna specifikationerna och lagrar sedan karaktärssträngen i det angivna fältet. Man kan nu alltså generera sina data i en godtycklig ordning - det behöver alltså ej genereras rad för rad. Den andra subrutinen skriver ut hela sidan och initialiserar sedan en ny med fix information.

15c. ACOM - Enkelt interaktivt beräkningsspråk

ACOM är ett trebetygsarbete utfört av Sven-Göran Jönsson. Målsättningen var att implementera ett enkelt beräkningsspråk, för att kunna användas under TSO. Då projektet påbörjades fanns ingen tillgång till ett sådant enkelt språk. Nu senare kan Basic användas.

Användaren har möjlighet att använda sig av variabler och matriser, och en beräkningssats kan innehålla de vanliga operatorerna. Exempel på satser:

37 + 45/3.456 ?	Beräknar och skriver ut resultatet.
ALFA = 123.45 * PI/3	Tilldelar variabeln ALFA värdet av uttrycket.
ALFA ?	Skriver ut variablen ALFA.
DIM '2,2'	Allokerar utrymme för en 2 X 2 matris.
ZETA=1 : 2.23	Tilldelar utrymmet namnet ZETA och initialiserar första kolonnens element till 1 resp. 2.23.
3.45 : 5;	Andra kolonnen tilldelas 3.45 resp 5.
ZETA'1,2' = ADAM	Tilldelar ett matriselement värde.
ZETA ?	Skriver ut matrisen ZETA.
ZETA = ZETA + ZETA	Matrisen ZETA's element fördubblas.

Programmet gör vissa felkontroller, såsom felaktig syntax, odefinierad variabel, kontroll av indexgränser m m. En begränsning i programmet är att man ej kan avallokera matrisutrymme, vare sig manuellt eller automatiskt med hjälp av en rutin för sophämtning ("garbage collection").

16. A.I.-språk

Forskning inom artificiell intelligens har ända sedan i slutet av 50-talet utvecklat och tillfört nya idéer för programmeringsspråk. Bearbetning av liststrukturer och symbolmanipulation har sina rötter i språk som IPL-V (1957) och LISP (1960). Den fortsatta utvecklingen av AI-forskningen, med studier i automatisk slutsatsdragning, representation av kunskap, naturligt språk, robotar och automatisk programmering, har emellertid känt behov att införa nya egenskaper och konstruktioner i dessa språk. Den nya Planner- idéen utvecklades vid MIT av Carl Hewitt (ref 1) med start 1967. Liknande idéer fanns även på flera andra håll, t ex LISP A som utvecklades i Uppsala 1967-68 (ref 2). Försök påbörjades att implementera Planner i assemblykod, men detta har övergivits, och i stället har påbyggnader av etablerade språk utförts. I LISP har Microplanner, Conniver, QA4 samt QLISP och i POP-2 har Poplar implementerats. Dessa nya konstruktioner påverkar även implementeringen av dessa underliggande språk och då främst LISP, där man för tillfället håller på med att införa nya kontrollstrukturer efter den modell (spagetti-stackar), som Bobrow och Wegbreit (ref 3) har utarbetat.

Vid den senaste AI-konferensen i Stanford hölls ett antal föredrag om denna nya klass av avancerade språk, och i (ref 4) ges en god sammanfattning.

Nu följer en beskrivning av några nya faciliteter. Dessa kan indelas i

- data typer
- kontrollstrukturer, inkluderande pseudo-parallellism, kontexter och backtrackning
- mönstermatchning, använd både för att hämta data och för programkontroll
- automatisk slutsatsdragning

Nya datatyper. Tidigare har man normalt haft tillgång till de symboliska datatyperna atom, lista och strängar tillsammans med de numeriska data-

typerna. Mer komplexa program har visat behovet av ytterligare datatyper och i QLISP har man exempelvis infört tupler, mängder, bagar (mängd med upprepning av element). Vissa problem uppstår vid implementeringen av dessa. Datatyperna måste överföras internt på kanonisk form och ha en unik lagrings-representation. Man kan även tänka sig användardefinierade datatyper, där användaren dessutom får skriva vissa elementära rutiner för I/O, evaluering, allokering och garbage av datatypen.

Kontrollstrukturer. I konventionella programmeringsspråk (FORTRAN, Algol) har man en mycket enkel kontrollstruktur. Kontrollen mellan moduler (subrutiner, funktioner, procedurer) är helt hierarkisk, såtillvida att återhopp från en modeul endast är tillåten till den anropande modulen. Rekursiva anrop är tillåtna i vissa av språken. Vid anrop av en modul sker normalt en bindning av variabelnamn och värden. För att klara rekursiviten skaps en instans av modulen. Retur från en modul medför att den försvinner och man har ingen möjlighet att "aktivera" den igen eller att komma åt dess variabelbindningar.

I LISP finns en sk. funarg-mekanism, vilket tillåter möjligheten att spara en instans av en modul tillsammans med ett antal variabler och dess aktuella bindningar. Denna instans kan sedan evalueras i en helt annan omgivning.

I SIMULA finns möjlighet att skapa modul-instanser (class), exekvera i den, avbryta och spara omgivningen för att överföra kontrollen till en annan instans. En modul behöver inte överföra kontrollen tillbaka till den anropande modulen utan kan överföra den till vilken annan modul som helst.

För att införa flexiblare kontrollstrukturer har en modell, spagetti-stackar, utarbetats (ref 3). Denna modell beskriver vilken information, som måste sparas tillsammans med varje modul-instans. Följande information ingår:

- binding link, som anger var värdena (bindningarna) till lokala variablerna finns till denna modul instans (LISP:s lambda- och prog- variabler).

- access link, som anger i vilken omgivning man söker efter globala variablers värden.
- control link, som specificerar vilken modul instans, som skall fortsätta bearbetningen efter normal "return".
- process tillstånd, som anger var och hur den fortsatta bearbetningen i denna modul skall påbörjas.

Med denna modell kan man nu implementera följande:

System av korutiner, där bara en process är aktiv åt gången, såsom i SIMULA, eller system av multiprocesser, där flera processer är aktiva och där en executiv process eller time-slice mekanism fördelar bearbetningstid..

Backtracking, Ett specialfall av korutiner, där modul-instansen sparas vid speciella besluts-punkter (SELECT-satser enlig Floyd terminologi (ref 5)) och återhopp till närmaste besluts-punkt om bearbetningen misslyckas (FAIL) vid senare stadium. I detta fall återställs kontrollstrukturen automatiskt. Ändringar i globala data, som utförts efter beslutspunkten måste "undos". I INTERLISP sker denna "undo" för all ändringar genom att bibehålla en historia över dessa global ändringar. I QLISP kallas detta att man logiskt kan arbeta i olika dataomgivningar, sk. "contexts".

Ytterligare avancerade strukturer kan skapas med ovanstående modell, genom att system tillhandahåller primitiver för att man själv skall kunna ändra de olika länkarna i en frame. I ett INTERLISP system finns motsvarande möjlighet att själv ändra i variabelstacken med de sk. stackfunktionerna.

Mönstermatchningen av datastrukturer infördes först i strängbearbetande språk som SNOBOL, men har visat sig mycket värdefull även för matchning av datastrukturer, och därmed i andra tillämpningar inom AI. Man har alltså möjlighet att komma åt data-objekt (som kan vara liststrukturer) genom att specificera ett mönster, som detta dataobjekt matchas emot.

I stora komplexa system med en mängd procedurer, kan det vara svårt för en användare att i varje steg exakt namnge de procedurer han vill anropa. Detta kan bero på att det kanske finns flera möjliga procedurer eller att procedurerna dynamiskt tillförs och tas bort från systemet. Till varje procedur associeras en mall. Anropet till en procedur består då i att ange ett mönster och sedan är det systemets uppgift att välja den eller de procedurer, som matchar mönstret. Systemet väljer en procedur i taget på ett icke-deterministiskt sätt, dvs den väljer en procedur och skulle den ej vara bra väljs nästa etc. Denna användning kallas mönsterstyrd invokering av procedurer, och tillåter en mer flexibel möjlighet att arbeta med stora komplex AI-program. Procedurer, exempelvis axiom som tillför systemet mer "kunskap", kan enklare integreras och behöver då normalt inte påverka tidigare program-moduler.

Kort beskrivning av QLISP (ref 5). Vid SRI utvecklades ett helt nytt språk QA4 (ref 7), innehållande många av de nya faciliteterna, som beskrivits ovan. Språket interpreterades av LISP, men blev mycket ineffektivt på grund av denna "dubbelinterpretering" och avsaknaden av att kunna kompilera QA4-satser till maskinkod. De viktigaste faciliteterna bröts ut och implementerades i INTERLISP, på så sätt att de var åtkomliga på LISP-nivån. Detta nya system kallas $_{Q}$LISP. För en användare av QLISP, som bara vill ha tillgång av LISP, kommer de nya faciliteterna att bli osynliga. Åtkomsten sker via LISP:s faulteval-mekanism.

Elementen av datatyperna class (mängder), bag (mängder med upprepning av element), vector (n-tupel) och tuple (lista) lagras alla i en permanent databas, kallad "discrimination net". Nya element överförs till kanonisk form, så att alla lika uttryck har identisk representation och plats i nätet. Elementen kan även, som en LISP-atom, ha tillgång till egenskapslista. Följande två uttryck är då identiska:

(CLASS ONION (BAG APPLE TOMATO APPLE)(VECTOR POTATO EGG))
(CLASS (BAG TOMATO APPLE APPLE) ONION (VECTOR POTATO EGG) ONION)

En variabel i QLISP kan förekomma i tre olika typer, där ett prefix anger typen.

←X X tillåts att tilldelas ett värde

?X X tillåts att tilldelas ett värde, endast under förutsättning att det ej har något värde tidigare

$X anger värdet av variabel X, som den tidigare måste ha tilldelats

En atom utan prefix, står för sig själv ("inverse quote mode"). Tilldelning av variabelvärden sker sedan via mönstermatchning med funktionen matchqq (analog med setqq)

```
(MATCHQQ(TUPLE (BAG ←X ←X ←Y)(TUPLE ←X $Z))
        (TUPLE (BAG ADAM BERTIL ADAM)(TUPLE ADAM CAESAR)))
```

Om nu Z tidigare har värdet CAESAR kommer X att få värdet ADAM och Y värdet BERTIL.
En funktion i QLISP har följande utseende:

(QLAMBDA mall funktionskropp),

där mall är ett QLISP-uttryck, och där funktionskroppen kan antingen bestå av godtyckliga LISP-satser eller speciella QLISP-satser, såsom qprog, qget, gput etc

En funktion foo, som överför listan (A (B C)) till ((C B) A) skrivs i LISP som

```
(LAMBDA (X)(LIST (LIST (CADADR X)(CAADR X))(CAR X)))
```

men i QLISP, den kanske mer lättförståeliga funktionen

```
(QLAMBDA (TUPLE ←X (TUPLE ←Y ←Z)) (TUPLE (TUPLE $Z $Y) $X))
```

Om vi försöker använda denna funktion på det felaktiga argumentet (A B C) kommer LISP-programmet att göra en feldiagnos på låg nivå eller kanske värre att returnera ett felaktigt och meningslöst resultat, medan QLISP-programmet omedelbart kommer att misslyckas med att matcha lista (TUPLE A B C) med mallen i QLAMBDA-uttrycket, och alltså svara "värde okänt", eller söka en annan definition av funktionen foo.

Man har möjlighet att arbeta med en databas i olika contexter, dvs man börjar med en global databas och vid övergång i nya contextar

kommer alla ändringar att vara lokala i denna context.

För att lagra ett uttryck i nätet finns funktionen qput (samt några varianter av denna, såsom assert, deny etc) och för hämtning funtionen qget (även där varianter, såsom is och goal).

```
(QPUT (FATHER KARL $X) APPLY $AXIOMS  WRT  CONTEXT5 IND1 PROP1)
```

Om nu X har värdet ADAM kommer följande att inträffa:

- (FATHER KARL ADAM) lagras i nätet
- $AXIOMS är namnet på en lista av funktioner. I detta fall kan funktionerna var framåtslutsatsdragningregler och de funktioner vars mall matchar (FATHER KARL ADAM) kommer att evalueras.
- All lagring sker i contexten CONTEXT5 (wrt står för "with respect to")
- På (FATHER KARL ADAM):s egenskapslista lagras egenskapen PROP1 under indikatorn INS1.

```
(QGET (←Z KARL ←Y) WRT CONTEXT5)
```

kommer att binda variablerna Z och Y till motsvarande element i första uttrycket, som systemet finner i denna context.

Med hjälp av funktionen bis kan man sätta upp en beslutspunkt, och söka efter uttryck med hjälp av backtrackning. Om det på lägre nivå inte finns något matchande uttryck eller om man explicit utför en fail, återgår kontrollen till närmaste bis-sats, som fortsätter bearbetningen.

Vid DLU skall under 1974 försök göras med att lägga över en version av QLISP, som vi fått tillgång till från SRI. Ansvarig för utvecklingen av QLISP på SRI har bland annat varit René Reboh, tidigare medlem av DLU, och som beräknas återkomma hit hösten 1974. Vi hoppas kunna etablera en grupp vid DLU för att studera dessa nya språk och nya faciliteter i dem. Vid DLU kallar vi dem för "ultraspråk" och för tillfället ägnas största intresset åt nya konstruktioner i påbyggnaderna på LISP, samt studier av SIMULA (se kapitel 6 i denna rapport). Många konstruktioner är ej riktigt testade vad gäller användbarheten, och några, bland annat backtracking, har missbrukats och därför kritiserats (ref 8). Användarna

har överanvänt ickedeterminismen, vilket förorsakat alltför ineffektiva program. Nya formalismer, såsom actors (ref 9), bör även studeras. Vid DLU planeras eventuellt att implementera spagetti-stackar och användardefinierade datatyper i det tillsammans med UDAC utvecklade INTERLISP--systemet för IBM 360/370.

Referenser

1. Carl Hewitt
DESCRIPTION AND THEORETICAL ANALYSIS (USING SCHEMATA) OF PLANNER: A language for proving theorems and manipulating models in a robot.
AI memo 251, MIT Project MAC, April 1972

2. Erik Sandewall
LISP A: A LISP-like system for incremental computing
Proc. 1968 Spring Joint Computer Conference

3. Daniel Bobrow & Ben Wegbreit
A MODEL AND STACK IMPLEMENTATION OF MULTIPLE ENVIRONMENTS
CACM, vol 16, nr 10, Oktober 1973

4. Daniel Bobrow och Bertram Raphael
New Programming Languages for AI Research
Tutorial Lecture at THIRD INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, Stanford University, Stanford,Aug 1973

5. Robert Floyd
Non-deterministic Algorithms
JACM, vol 14, nr 4, okt 1967

6. Rene Reboh och Earl Sacerdotti
A PRELIMINARY QLISP MANUAL
SRI, Stanford, Aug 1973

7. J. F. Rulifson et al.
QA4, A LANGUAGE FOR WRITING PROBLEM-SOLVING PROGRAMS
Proceeding IFIP Congress, 1968

8. Gerald Sussman och D. W. McDermott
WHY CONIVING IS BETTER THAN PLANNING
AI memo 255A, MIT Project MAC, April 72

9. Carl Hewitt et al.
A UNIVERSAL MODULAR ACTOR FORMALISM FOR ARTIFICIAL INTELLIGENCE
Proc. IJCAI, Stanford, Aug 1973

ARBETSFÖRDELNING OCH SPONSORER

Här följer en uppställning som motsvarar innehållsförteckningen, vari angivits vilka som arbetat med uppgiften, samt anslagsgivare eller motsvarande. Uppställningen avser endast 1973, ej tidigare år och ej planer för 1974. Använda förkortningar förklaras sist.

1. Allmänt om verksamheten vid DLU		
2. Bakgrund och översikt (programmeringssystem och -språk)		
3. Maskinnära system		
3a. Studium av implementeringsspråk	Anders Beckman	STU
3b. Kcrsassembler för Nova	Åke Malmberg	STU
4. Programmeringssystem för smådatabaser		
4a. INTERLISP/370	Jaak Urmi	UDAC
4b. Studium av hashningsmetoder	Lars Lidén	3-betygs-arbete
4c. NOVALISP	Åke Malmberg	STU
4d. Effektivitetsmätningar i LISP F1	Mats Nordström	FOA P
4e. LISP F1 och F2	Mats Nordström	FOA P
4f. PLAST	Olle Willén	FOA P
4g. REC	Torgny Tholerus	-
5. Manipulation av FORTRAN-program		
5a. FFC (=Fortran-to-Fortran Converter), slutrapport	Anders Beckman	STU
5b. FAP (=Fortran Analysis Program)	Lars Göhran	3-betygs-arbete
5c. REFLIST (referenslistegenerator för Fortran)	Lars Göhran	STU
5d. Tidsanalysprogram för Fortran	Bo Elofsson	3-betygs-arbete
5e. Standard-Fortran-verifikator	Tom Smedsaas	-
6. Teori för programmeringsspråk		
6a. SIMLISP	Mats Nordström	FOA P
6b. VDL-interpretator	Mats Nordström	FOA P

7. Bakgrund och översikt (programredskap och metoder)

8. Manipulation av LISP-program

8a. REDFUN och REDCOMPILE	Lennart Beckman Östen Oskarsson	STU
8b. REMREC: prestandamätningar	Tore Risch	RJF
8c. MAKRO	Tore Risch	RJF
8d. PMG	Tore Risch	RJF
8e. Funktionslexikon	Lennart Beckman	STU

9. Programredskap för informationsstrukturer på lägre nivå

9a. PCDB	Anders Haraldson	STU
9b. GIP/GUP	Östen Oskarsson	STU
9c. TOP	Torgny Tholerus Mats Nordström	FOA
9d. TABLEOUT	Lennart Beckman	STU

10. Programredskap och metoder på naturligt-språk-nivå

10a. PCF-2	Erik Sandewall	UU
10b. SEPAC	Erik Sandewall	UU
10c. Nätverksparser o -grammatik	Mats Cedvall	NFR
10d. FORIN (formelmeningar)	Erik Sandewall	UU

11. Bakgrund och översikt (användningar av smådatabaser)

11a. Översikt över möjliga tillämpningar

12. Tillämpningar och experiment

12a. CICERON	Sture Hägglund	IBM
12b. DLUORG	Olle Willén	STU
12c. Frågesystem för naturvårdsbas	Sture Hägglund	IBM
12d. HOUSE (PCDB-experiment)	Christer Eck	3-betygsarbete

13. Användningar i annan forskning

13a. ATMAN	Hans-Jürgen Holstein Tore Risch	RJF

14.	Naturligt-språk-arbete		
	14a. Konversationsmodell	Mats Cedvall	NFR
	14b. Disambiguering	Erik Sandewall	UU
	14c. Programsammankoppling	Mats Cedvall	NFR
15.	Övrigt		
	15a. Test av PPS-metoden för heuristisk sökning	Lars Wahlgren	3-betygs-arbete
	15b. Helsidesformat i Fortran	Kai Sippel	3-betygs-arbete
	15c. ACOM	Sven-Göran Jönsson	3-betygs-arbete
16.	A.I.-språk	Anders Haraldson	

Använda förkortningar

FOA P	Försvarets forskningsanstalt, avdelning P
IBM	IBM:s FUSAM-kommitté (Forskning-Undervisning-Samverkan)
NFR	Naturvetenskapliga forskningsrådet
RJF	Riksbankens Jubileumsfond
STU	Styrelsen för teknisk utveckling
UDAC	Uppsala datacentral
UU	Uppsala universitet

När anslagsgivare markerats med -, har arbetet gjorts utan viss anslagsgivare och med marginalresurser.

TIDIGARE ÅRSRAPPORTER FRÅN DLU

1. Artificiell intelligens (1970), utkom januari 1970
2. Datalogilaboratoriet (1971), utkom januari 1971
3. Datalogilaboratoriet (1972), utkom januari 1972
4. Datalogilaboratoriet (1973), utkom januari 1973

RAPPORTER OCH DOKUMENTATION FRÅN DLU UNDER JANUARI - DECEMBER 1973*

DLU 73/1 Erik Sandewall: Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs.

DLU 73/2 Olle Willén: Dokumentation av den experimentella implementationen PLAST, mars 73, CS report no 46

DLU 73/3 Olle Willén: PLAST: Formell definition, 73-03-23

DLU 73/4 Mats Nordström: Synpunkter på VDL sett från LISP, 73-04-09

DLU 73/5 Erik Sandewall: Tillämpningar av smådatabassystem - diskussionsunderlag, 73-04-24

DLU 73/6 Erik Sandewall: Documentation of the SEPAC package, 73-05-02

DLU 73/7 Erik Sandewall: Documentation of the SEPAC data structure, 73-03-24

DLU 73/8 Anders Haraldson: Programdokumentation CS2L, 73-06-07

DLU 73/9 Anders Haraldson, Erik Sandewall: PCDB System Documentation, Part I, 73-06-21

DLU 73/10 Ingvar Kvarbäck: Testexempel till PCDB, 73-07-24

DLU 73/11 Olle Willén: DLUORG - Beskrivning av Datalogilaboratoriets interna organisation, 73-07-24

DLU 73/12 Kai Sippel: Helsidesformat i FORTRAN - ett program för generering av sidouppställningen i ett Fortran-program, 73-01-17

DLU 73/13 Olle Willén: Utvärdering av enkätundersökning angående smådatabaser och PLAST, 73-07-25

DLU 73/14 Lars Göhran: FAP - ett program för analys och syntes av Fortran-rutiner, 73-06-07

DLU 73/15 Bo Österberg: PACO - ett program för interaktiv parametermodifikation på Siemens 305, 73-06-25

---

* En fullständig förteckning för tiden 1966-73 är tillgänglig på begäran.

DLU 73/16 Arne Svennson: FFCMAC - en utveckling och implementering av macro-språk för Fortran i FFC, 73-05-15

DLU 73/17 Östen Oskarsson: Programdokumentation GIP/GUP, Generell in- och utmatning av egenskapslistor, 73-07-05

DLU 73/18 Mats Nordström: TOP - en parser för programmeringsspråk, 73-09-28

DLU 73/19 Sven-Göran Jönsson: ACOM: ett nytt program för att utföra beräkningar med enkelt språk via tso-terminalen, 73-08-13

DLU 73/20 Sture Hägglund: INSTANT CICERON - CICERON kort handledning, 73-10-05

DLU 73/21 Anders Beckman: Existerande programmeringshjälpmedel: DDT för DEC-10, 73-10-29

DLU 73/22 Tore Risch: MAKROEXPAND - ett enkelt makroexpanderingspaket i LISP

DLU 73/23 Anders Haraldson: PCDB - A Program which Generates Procedures for Maintaining a Data Base of Formulas in Predicate Calculus, 73-11-13

DLU 73/24 Tore Risch: REMREC - A program for automatic recursion removal in LISP, 73-11-12

DLU 73/25 Olle Willén: IKP (Infix and Keyword Parser), 73-11-14

DLU 73/26 Sture Hägglund: The LISP editor

DLU 73/27 Erik Sandewall: Some examples of disambiguation through deduction, 73-11-12

DLU 73/28 Tore Risch: REMREC - Ändra rekursiv LISP-kod till icke-rekursiv sådan, jan 1973

DLU 73/29 Lars Wahlgren: Lösning av ett 15-spel med PPS-metoden, 73-01-15

DLU 73/30 Stig Holmberg: Garbage collectionfri minnesdel i LISP F1 samt överföring av commonblocket mellan kärnminne och skivminne, 73-03-01

DLU 73/31 Anders Beckman: Synpunkter på Fortran-kompilator till projekt P, 73-06-12

DLU 73/32 Anders Beckman: Implementering av "funktionsknappar", 73-08-13