DLU 75/A1

DATALOGI-
LABORATORIET
(1975)

En beskrivning av
forskningen under 1974
och planerna för 1975

UPPSALA UNIVERSITY
Datalogilaboratoriet
Department of Computer Sciences

ISBN 91-506-0005-2

DATALOGILABORATORIET 1975-01-24 DLU 75/A1

Inst f informationsbehandling
Uppsala universitet

D A T A L O G I -
L A B O R A T O R I E T
(1 9 7 5)

En beskrivning av forskningen vid Datalogilaboratoriet under 1974 och av planerna för 1975, skriven av Anders Beckman, Mats Cedvall, Pär Emanuelsson, Anders Haraldson, Hans-Jürgen Holstein, Sture Hägglund, Klas Mårtensson, Mats Nordström, Östen Oskarsson, René Reboh, Tore Risch, Erik Sandewall, Torgny Tholerus, Jerker Wilander och Olle Willén. I forskningsarbetet har även deltagit Lennart Beckman och Jaak Urmi.

ISBN 91-506-0005-2
Uppsala 1975

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 15 personer på heller 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 1974 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 1975/76.

I detta inledande avsnitt skall vi på ett fåtal sidor först beskriva målsättningen av arbetet under 1974. Målsättning och uppläggning är i stort identisk med den som beskrevs i föregående årsrapport.

---

* Detta kapitel har översiktskaraktär, och är ett något reviderat utdrag ur motsvarande kapitel i föregående årsrapport (DLU -74).

** 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 årsrapporten (DLU -73).

Det har varit vår avsikt att denna årsrapport skall kunna läsas oberoende av tidigare årsrapporter. Därför är vissa avsnitt i stor utsträckning upprepningar av motsvanda i förra årsrapporten.

Definition av forskningsområde. Verksamheten vid DLU har till största delen inriktats på metodproblem vid intensivbearbetning av små databaser, varvid bl a resultat från forskningen i artificiell intelligens kommer till användning. En utvidgning av verksamheten har skett genom en grupp, som arbetar på tillförlitlighet av programmeringssystem. Annan verksamhet, med större eller mindre anknytning till små databaser, är systemprogrammering och studium av den semantiska definitionen av SIMULA. Nedan specificeras några av dessa områden närmare.

Små databaser.* 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. Exemple: 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

---

* Detta avsnitt är identiskt med motsvarande i DLU -74. En rapport "Små databaser" har sammanställts med utdrag ur tidigare årsrapporter, där området definieras och olika tillämpningar beskrivs. Denna rapport kan beställas från Datalogilaboratoriet, Sturegatan 1, 752 23 Uppsala.

- 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 referenser 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 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.

---

* Ej att förväkla 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.

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 verca, 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. Förekommande 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 småbasproblemen 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 och manipulationerna i databasen?
interaktion	hur välja en lämplig notation för kommunikation med systemet?	hur översätta mellan extern och intern notation?
	hur ställa upp delar av databasen i tabellform för presentation?	

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 rest. 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.

Tillförlitlighet i programmeringssystem. Under de sista åren har tillförlitligheten hos programvara blivit ett alltmer aktuellt problem. Detta beror delvis på att allt större programsystem utvecklats men också på att datorer används för allt mer kvalificerade uppgifter som styrning av tillverkningsprocesser, styrning av ställverk och hantering av mycket stora databaser där mänsklig kontroll av alla enskilda poster är nästan omöjlig. Programvarufel har betraktats som ofrånkomliga och testningen av kritiska program har kommit att kosta lika mycket som programutvecklingen i övrigt.

Ett av resultaten av det tidigare projektet "totala programmeringssystem" var att det inte var möjligt att med understödsprogram etc höja tillförlitligheten hos dagens konventionella programmeringssystem till en acceptabel nivå eftersom de har en "inbyggd" otillförlitlighet som inte kan kompenseras bort. Detta ledde till ett djupare studium av problemet tillförlitlighet.

Vid utvecklingen av ett program tar man först fram en specifikation som beskriver vad programmet skall utföra. Sedan tar man fram algoritmer som beskriver hur detta skall utföras, specificerar delproblem och väljer algoritmer för dessa. Nästa steg är att överföra algoritmerna till ett programmeringsspråk. Det så erhållna programmet översätts sedan till maskinkod som slutligen exekveras på målmaskinen. Om ett program skall kunna vara tillförlitligt måste varje steg i ovannämnda process vara tillförlitligt. Det räcker inte att förbättra ett av stegen, t ex programmeringsspråket, om det finns andra steg som är avsevärt sämre.

I varje steg finns möjligheter till förbättringar. Specifikationen kan formaliseras så att relationerna nellan element i specifikationen och programmet kan uttryckas. Mycket arbete pågår för att finna metoder för att bevisa algoritmers korrekthet. Utvecklingen av programmeringsspråk har länge varit ett populärt område där trots detta mycket kvarstår att göra. Maskinvaran kan göra exekveringen säkrare genom t ex typmärkta data avancerade skyddssystem och mikrokodade programstruktureringsprimitiver. Dessutom kan hjälpmedel för testning och felsökning i program göras avsevärt bättre än de som nu finns.

Utgångspunkter för arbetet med programvarutillförlitlighet vid DLU är att programmeringsmetodiken, som bestämmer hur de olika stegen i utvecklingen utförs, skall vara avgörande för hur programmeringsspråk, maskinarkitektur etc bör utformas.

Former för verksamheten under 1974. 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:

Carbonell Memorial Symposium vid Pajaro Dunes (Sandewall).

International Summer Seminar on Concepts of Automatic Processing of Natural Language (Cedvall)

IFIP 74 (Beckman, Haraldson, Mårtensson, Reboh, Sandewall, Urmi)

Under året har Datalogilaboratoriet gästats av följande föreläsare:

Prof A.P. Ershov (Novosibirsk)
Dr Carl Hewitt (MIT)
Dr Robert Kowalski (Edinburgh
Dr S. Machonin (Institute for Information Transmission Problems, (Moskva)
Dr Bert Raphael (Wien och Stanford Research Institute)
Dr Warren Teitelman (Xerox Palo Alto Research Center)
Dr Richard Waldinger (Stanford Research Institute)

Följande personer har under året vistats en längre tid vid Datalogilaboratoriet:

Dr S. Machonin (Institute for Information Transmission Problems, Moskva) - under 6 veckor för studier i artificiell intelligens.

Dieter Kolb (Institute für Deutsche Sprache) - under 2 veckor för att lära INTERLISP.

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 (Huvudenhet 1), IBM Svenska AB, Naturvetenskapliga forskningsrådet, Riksbankens Jubileumsfond Institutet för tillämpad matematik, Styrelsen för teknisk utveckling samt Sekretariatet för nordiskt kulturellt samarbete. 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
Göteborgs universitet (Institutionen för informationsbehandling)
IBM Svenska Aktiebolag
KTH (Institutionen för informationsbehandling och Institutionen för Teletrafiksystem)
L M Eriksson
Statistiska centralbyrån (enheten för databehandlingsmetodik)

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

2. KURSVERKSAMHET

En del av verksamheten vid DLU är att sprida resultat från den pågående forskningen genom kursverksamhet. Detta görs i form av doktorandkurser, endagsseminarier, kurs hos företag, sommarskolor etc. Detta avsnitt redogör för kursverksamheten 1974.

2a. Doktorandkurser i datalogi vid institutionen för informationsbehandling vid Uppsala universitet

Små databaser. En kurs främst för att ge deltagarna möjlighet att testa och köra olika programredskap, som utvecklats för små databaser. Där ingick programmen Redfun, Redcompile, PCDB, Top, PMG, Nätverksparsen, GIP/GUP, DABA, samt några smärre program. Kursen var upplagd, så att varje program presenterades, mest med avseende på praktiska problem att använda programmet, och uppgifter delades ut. Efter det att uppgiften var löst diskuterades och kritiserades programmet av kursdeltagarna. Kursledare var Erik Sandewall.

AI-språk. I kursen presenterades idéerna i AI-språken och avsnitt om backtracking, mönstermatching, kontexter och diskrimineringsdatabas genomgicks i detalj. Sedan presenterades språket QLISP och övningsuppgifter utdelades. Föreläsare för kursen var René Reboh.

Kompilatorteknik. I kursen genomgicks grundläggande metoder och algoritmer som används i en kompilator, såsom scanning, parsing och kodgenering. Under kursens gång utdelades ett antal övningsuppgifter som anslöt till genomgångna algoritmer. Kurslitteratur var D. Gries: Compiler Construction for Digital Computers (Wiley 1971). Föreläsare var Anders Beckman.

Operativsystem. En kurs om idéerna i ett operativsystem avslutades under VT 74. I kursen diskuterades bl a problemen vid parallelism mellan processer, resursallokering virtuellt minne och protection. Kurslitteratur var aktuella artiklar från tidskrifter där grunden bildades av P.Dennings översiktsartiklar i Computing Surveys. Föreläsare var Jaak Urmi.

2b. Forskarkurs i datalogi, med inriktning små databaser.

Under två veckor i juni 1974 arrangerade DLU en sommarskola för forskarstuderande i informationsbehandling med datalogiinriktning, med deltagare från de nordiska länderna. Kursen bekostades av Nordiska Forskarkurser.

Deltagare kom från Danmark, 3 st; Finland, 4 st; Norge, 10 st; samt Sverige 13 st. Sammansättningen av gruppen var att deltagarna kom från hela informationsbehandlingsområdet, dvs både ADB, numerisk analys och datalogi. Denna skillnad i förkunskaper påverkade inte i hög grad förståelsen av det genomgångna materialet. Däremot var det en viss skillnad på vanan att handha terminal och att utföra programmering interaktivt.

Programmet bestod av föreläsningar på förmiddagar och laborationer på eftermiddagarna. Vissa kvällar användes för seminarier. Innehållet på kursen var att presentera den forskning på små databaser vi utför vid DLU. Följande kursmoment ingick

- Programmeringsspråket LISP
- Små databaser. Problempresentation, datarepresentation, sökmetoder och andra programvaruhjälpmedel.
- Språk för kommunikation med liten databas. Programmeringsmetoder och olika typer av parserprogram.
- Genomgång av CICERON (se DLU/ 72).
- Utvecklingstendenser, nya typer av språk.

Laborationerna bestod av att på dator själv utveckla mindre program samt att testa och i vissa fall modifiera existerande smådatabasprogram. Under laborationerna hade vi tillgång till UDAC's IBM/370 och QZ's dialogdator Simon. På UDAC körde vi INTERLISP och på Simon använde vi en INTERLISP-simulator (se 9c) för dess LISP 1.6. Föreläsningarna utfördes av Mats Cedvall, Anders Haraldson, Sture Hägglund, Mats Nordström, Erik Sandewall och Jaak Urmi. Dessa plus övriga vid DLU tjänstgjorde som grupp- och handledare. Seminarier hölls dessutom av Jacob Palme, FOA och Olle Willén.

Det allmänna intrycket, både från deltagarna och lärarna, var att sommarskolan blev mycket lyckad och uppfyllde sitt syfte, som forskarkurs. I en sammanfattningsfråga ur en enkät om hur kursen i sin helhet varit erhölls resultaten

18 mycket värdefull
6 ganska värdefull
1 föga värdefull
0 helt värdelös

Kurslitteratur och viss dokumentation från laborationerna finns fortfarande att tillgå mot en mindre avgift från Datalogilaboratoriet.

För genomförandet av sommarskolan tackar vi förutom Nordiska Forskarkurser, även institutionen för informationsbehandling vid KTH för lån av ett antal terminaler, samt FOA och UDAC som ställt maskintid till vårt förfogande på Simon resp IBM/370.

2c. Övrigt

Vid en sommarskola i Pisa, Italien om "Linguistic Data Processing", höll Erik Sandewall en kurs i logik.

Hos LM Eriksson har Mats Nordström hållit en veckolång kurs i LISP. Exemplen på kursen var tagna från verkliga problem hos LM Eriksson.

P R O G R A M M E R I N G S S Y S T E M O C H – S P R Å K

3. 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.

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 (Siemans 305) och en där vi inte var nöjda med existerande implementation (IBM 360), samt ett portabelt system skrivet i Fortran, kallat LISP F1. Vidare göres arbete på olika "överbyggnader" av LISP-systemet, samt grundläggande arbete på (maskinnära) implementeringsspråk.

LISP:s principer för uppbygnad av programmeringssystem är mycket intressanta, men genom den (åtminstone vid första påseende) obekväma notationen

---

* Detta kapitel har översiktskaraktär, och är ett något reviderat utdrag ur motsvarande kapitel i föregående årsrapport (DLU-74).

i språket LISP torde detta knappast komma att nå någon större spridning. Vi försöker ändå föra ut resultat från vårt arbete på LISP-system bl a genom att förbättra LISP:s notation, så att den blir bekvämare speciellt för nybörjaren. Den resulterande notationen ytspråket kallas PLAST och beskrivs i kapitel 4b.

Utöver PLAST har också annat arbete bedrivits på vidareutveckling av programmeringsspråk. Här ingår dels en formell definition av semantiken i Algol 60 och Simula (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. Den senare har resulterat i språket REC (se kapitel 4g). Det skall dock samtidigt påpekas att i stort sett att smådatabas-programmering vid DLU görs i LISP och att övergång till att använda en efterträdare till LISP troligen ligger långt fram i tiden.

Dessutom bedrivs arbete inom området tillförlitlighet i programmeringssystem. Detta arbete är inte inriktat speciellt mot LISP-system utan mot det generella problemet med tillförlitlighet. En utgångspunkt för arbetet är att programmeringsmetodiken skall bestämma utseendet hos programmeringsspråk, maskinarkitektur, hjälpmedel etc. Arbete är därför till stor del inriktat på programmeringsmetoder.

Slutligen har arbetet på hjälpmedel för Fortranprogrammering fortsatts. De analyserande programmen har kompletterats med en referenslistegenerator för COMMON-variabler. I anknytning till arbetet med tillförlitlighet i programmeringssystem har också utvecklats en språkutvidgning och en förprocessor som bl a innehåller de kontrollstrukturer som anses nödvändiga för välstrukturerade program.

3b. 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 m m har vi dock föredragit att göra en uppdelning så att endast a-c kallas programmeringssystem och behandlas i kapitel 3-7 av denna rapport, medan d-e räknas till programhjälpmedel och ingår i kapitel 9 och 10.

4. PROGRAMMERINGSSYSTEM FÖR SMÅDATABASER

Det primära programmeringsspråket vid Datalogilaboratoriet är LISP, vilket ger en gemensam grund för verksamheten. Härigenom har också implementering av LISP-system kommit att ingå i arbetet. Under 1974 har arbetet på ett system i dialekten INTERLISP för IBM 360/370 i stort slutfört. Systemet har i olika pre-releaser använts av olika personer vid DLU under större delen av 1974 och visat sig vara mycket pålitligt och lätt att arbeta med. Systemet har nu även installerats på olika maskiner uti Europa och används nu där i produktion. Vårt portabla LISP-system, LISP F1(F2) håller kontinuerligt på att skickas ut till olika användare i världen och den används också då vi vill utföra demonstrationer hos någon användare, som inte har LISP på sin maskin. I Sverige har LM Ericsson systemet och experiment håller på att utföras på det (se 13f).

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 litteraturen 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.

Under året har vi från SRI tagit över och implementerad QLISP på INTERLISP/360. Vi kommer att utföra ytterligare experiment med systemet. Vi ser nog dessa språk mer som en samling programredskap än som ett nytt programmeringsspråk. Under 1975 kommer vi att försöka isolera olika delar av QLISP till olika självständiga programpaket. På så sätt kan vi få en mönstermatchare, ett paket för hantering av diskrimineringsdatabas etc.

4a. INTERLISP/360-370

Implementeringen är nu i stort avslutad. Under året har arbete bland annat utförts på garbage collectorn samt för underhåll av systemet. Dokumenta-

tion har utarbetats i form av en manual INTERLISP REFERENCE MANUAL* och i form av ett undervisningsmaterial LISP - details* lämpat för den som vill lära sig INTERLISP/360-370.

4b. PLAST

Sedan arbetet med den allra första, experimentella versionen av PLAST avslutades för drygt ett år sedan, har verksamheten varit inriktad på två nya implementeringar av språket. Den väsentligaste av dessa måste sägas vara den som har gjorts på QZ:s DEC 10-maskin, och skälet till detta är framför allt dessa: Dels tillhandahåller denna maskin den rätta omgivningen för ett språk av PLAST:s typ, nämligen en starkt interaktiv miljö, vilket är nödvändigt för effektiv programutveckling. Dels är det förhoppningsvis lättare att nå ett större antal externa (försöks-) användare den vägen.

Att PLAST parallellt också implementeras på UDAC:s IBM 370-maskin har två orsaker: För det första måste de användare, som inte har tillgång till QZ, t ex de tre-betygsstuderande som får undervisning i PLAST, också ha möjlighet att pröva på språket. Den andra anledningen är att LISP-systemet på UDAC, INTERLISP, är ett "bättre" implementeringsspråk för PLAST än QZ:S Stanford-LISP, och förhoppningsvis kommer INTERLISP så småningom att nå större spridning, kanske också till QZ. Om detta sker, är i så fall ett redan uttestat och "färdigt" PLAST-system redan tillgängligt.

De båda PLAST-systemen har nu (jan -75) i huvudsak samma status. En del mindre skillnader förekommer, av vilka de flesta troligen är ganska enkla att sudda ut, medan andra är mer beroende på det underliggande LISP-systemet, och kanske därför måstetolereras.Den avgörande skillnaden ligger dock i det faktum, att eftersom INTERLISP inte finns tillgängligt interaktivt i tillräcklig utsträckning, så är den där implementerade PLASTen

---

* INTERLISP REFERENCE MANUAL kan beställas från UDAC, Box 2103, 750 02 Uppsala.
LISP - Details, kan beställas från Datalogilaboratoratoriet, Sturegatan 1, 752 23 Uppsala

att betakta som batchorienterad och körs också endast i batch. Övergången till interaktivitet vållar emellertid inte några större problem för PLAST:s del.

Språket PLAST

Arbetet med de båda implementeringarna har främst varit inriktad på att få dem jämbördiga, kompatibla. Framför allt är det INTERLISP-PLASTen som uppgraduerats, eftersom dess utveckling för ett år sen inte nått riktigt lika långt. Kodningen av en ny inputrutin och behandlingen av arrayer kan sägas vara det som vållat mest huvudbry. Dessutom har de båda systemen parallellt utökats med ett antal aritmetiska funktioner och med ett kommando för att sätta prioriteter på användardefinierade operatorer. Detta senare är viktigt sett ur aspekten "syntaxutvidgning". En del arbete har också lagts ner på att definiera om vissa delar av översättaren i INTERLISP-PLAST för att kunna kompilera PLAST-definierade rutiner. Det har ännu inte gjorts några större och metodiska försök till tidsjämförelse mellan okompilerad och kompilerad kod, men en grov uppskattning sätter tidsvinsten till någonstans runt 40%. Detta inkluderar då ett "förkompilerings"-steg, som i princip "rätar ut vägen" mellan den lispifierade PLAST-koden och LISP-interpretatorn. Man kan slippa tidsödande "dubbelinterpretering" genom att t ex ersätta ett anrop till en PLAST-evalueringsfunktion med ett anrop direkt till grundsystemets motsvarighet. Ytterligare försök till uppsnabbning och större tidsmätningar skall göras.

Sedan på detta sätt språket PLAST nått en tillfredsställande nivå och tillförlitlighet, har arbetet tagit en litet annorlunda riktning. Ett rimligt krav på ett programmeringsspråk eller -system är, att det skall tillhandahålla olika hjälpmedel för en användares uttestning och organisation av program. Två sådana hjälpmedel har implementerats; en editor för modifikation av PLAST-procedurer och ett enkelt filhanteringssystem (ref 1).

PIPE(X), PLAST Interactive Procedure Editor (X-perimental version).

Editorn är en integrerad del av PLAST-systemet och kan anropas dynamiskt under en körning. Den är avsedd för editering av procedurer, dvs alla

former av PLAST-rutiner. Editeringen sker direkt i kärnminnet i termer av språket PLAST (naturligtvis), och några externa filer behövs inte vid editeringen.

Med ett kommando av traditionell PLAST-typ (: EDIT procname), anropas editorn. Väl inne i denna styrs editeringen av enkla (ev enbokstaviga) edit-kommandon. Editering kan ske enligt två principer:

1. PLAST är ett radorienterat språk, och den enklaste formen av editering är i enlighet därmed radeditering, dvs utbyte, tillägg eller avlägsnande av hela rader i procedurtexten.

2. En något mer avancerad form av editering uppnås med möjligheten att byta ut, lägga till eller avlägsna vissa följder av symboler i procedurtexten. Ett mönster (= följd av symboler) anges som argument till ett edit-kommando, och vid den punkt i proceduren där mönstret hittas genomförs editeringen. (Denna metod bör inte förväxlas med teckeneditering, där det ju är följder av karaktärer som bygger upp ett mönster.)

Båda dessa former av editering utförs med samma kommandon; endast argumentens utseende skiljer sig åt. Tillgängliga kommandon för dessa typer av ändringar är REPLACE, INSERT och DELETE, som också kan skrivas på kortformerna R, I resp D. För att underlätta editeringsarbetet kan kommandona också föregås av en repetitionsfaktor.

Förutom ett par typer av list-kommandon innehåller editorn också en enkel användarstyrd mekanism för återställning av felaktigt gjorda ändringar.

Betraktad i sin helhet är editorn ett ganska litet och enkelt system styrd av endast ett fåtal kommandon. Den fyller emellertid även i detta elementära utförande en väsentlig uppgift som ett nödvändigt programmeringshjälpmedel. Alla de ändringar som kan tänkas behöva bli gjorda i en procedur är möjliga att utföra.

Emellertid är editorn åtminstone tills vidare att betrakta som ett experiment eller en skiss, och fortsatt försöksverksamhet får avgöra vilka modifikationer som måste göras.

Hittills har editorn implementerats endast i Stanfordlispen på DEC-10:an. Frånvaron av interaktiv tillgång till INTERLISP gör inte en editor av detta slag speciellt användbar, varför implementationen där får anstå till dess att detta krav uppfylls. Utvecklingen av editorn inskränker sig då till ett enkelt konverteringsarbete.

Filhantering i PLAST

Ytterligare ett nödvändigt programmeringshjälpmedel är möjligheterna att kunna spara procedurdefinitioner och data på yttre filer. I PLAST har nu implementerats faciliteter för detta, vilka till sina huvudsakliga delar överensstämmer med metoderna i INTERLISP.

Filer genereras med ett kommando ⁜ SAVE filnamn , vars utförande styrs av två globala listor definierade av användaren. Dessa listor innehåller information om vilka procedurer resp vilka datastrukturer som skall sparas på filen. Alla typer av procedurer kan tas om hand, och av data kan globala värden och egenskapslistedata sparas.

En fil laddas med kommandot ⁜ LOAD filnamn , varvid även de globala styrvariablerna återskapas.

Hela denna filhanteringsmetod är uppbyggd kring begreppet "generationer" av filer. För varje filgenerering med ⁜ SAVE filnamn skapas en ny generation av filen filnamn. Detta förfaringssätt är väsentligt ur back-up-synpunkt. På motsvarande sätt hämtar ⁜ LOAD filnamn den senaste versionen av filen om inte annat (med hjälp av ett generationsnummer) explicit anges.

När det gäller implementationen av filhanteringen i INTERLISP har dess rutiner för detta kunnat användas nästan helt och hållet. I detta system är också hanteringen av generationsnummer redan inbyggd, och vållar inga bekymmer. I Stanfordlisp, som saknar motsvarighet, blir hanteringen något besvärligare och paketet också något vidlyftigare. Administrationen av generationsnummer kräver t ex en extra hjälpfil. Användarens initialverksamhet blir pga systemskillnaderna något olika: I INTERLISP-PLAST:en skall han allokera ett partitionerat dataset för sina filer, i Stanfordlisp-PLAST måste han allokera utrymme för hjälpfilen.

Ett par funktioner som styr inläsningen och bestämmer varifrån, från användarens terminal eller från en yttre fil, input för PLAST:s READ-sats skall komma, har också implementerats.

Från programmeringsspråk till programmeringssystem

PLAST som programmeringsspråk kan alltså vid det här laget sägas vara fixerat till sin utformning. Vissa ytterligare utvidgningar och extra faciliteter (och naturligtvis korrektioner) kan kanske tänkas bli utförda, men språkets syntax och dess övergripande idéer kommer att behållas i sin nuvarande form. För språket PLAST:s vidkommande innebär detta, att arbetet bör inriktas på debugging, uppsnyggning och effektivisering.

I och med att grunderna för ett filhanteringssystem och en editor nu har implementerats, börjar man kunna skönja en utveckling mot ett PLAST-system.

De närmaste planerna är nu att utöka detta system med ytterligare moduler. Det som i första hand tilldrar sig vårt intresse är ett system för hantering av (små) databaser utanpå de möjligheter som språket redan erbjuder. Små försök i denna riktning har gjorts under hösten, och idéer och problem har diskuterats.

Databashantering

En liten databas i PLAST:s mening ska kunna bestå av (komplexa) datastrukturer i form av t ex träd, nätverk etc. Sådana strukturer får byggas upp av t ex records eller egenskapsvärden. Men man vill också till databasen gärna räkna sådana procedurer, som är avsedda att manipulera just denna specifika databas. Det kan då vara "skräddarsydda" procedurer för sökning i databasen eller för presentation av den.

En sådan databas kan, enligt vad diskussionerna visat, kunna genereras på två sätt:

1. Man anropar ett speciellt databasgenereringsprogram. Detta är ett promptande program, som själv ställer nödvändiga frågor till användaren. Det kan t ex gälla prototyper för recordklasser,

definitioner av relationstyper och strukturer och annat som är karaktäristiskt för en databas. Informationen skall vara tillräcklig för att systemet med hjälp av den skall kunna generera vissa specialrutiner för manipulation med denna specifika databas. Så småningom skall naturligtvis också data kunna matas in. Under dessa kontrollerade former kan då också data hela tiden kontrolleras mot de definierade strukturerna.

Program eller system som redan utvecklats vid DLU, t ex GIP/GUP för in- och utmatning av egenskapslisteinformation och PCDB för definition av relationer och strukturer, bör studeras i detta sammanhang, och delar av dessa kan kanske integreras i vårt program.

2. En databas kan också tänkas experimenteras, laboreras fram. Detta sker då inom ramen för själva språket PLAST, och kan vara ett nödvändigt förfaringssätt, när vissa karaktäristika hos den tilltänkta databasen inte på förhand är kända. Man bygger upp strukturer på "känn" utan riktigt ha klart för sig hur slutresultatet skall se ut. För att åstadkomma detta, måste det redan befintliga PLAST-systemet modifieras på vissa punkter, så att t ex alla data automatiskt "samlas ihop". Detta gäller åtminstone egenskapsvärden och records samt definitioner av recordklasser. Slutligen bestämmer sig användaren för att den struktur han genererat skall utgöra en databas, och då måste systemet hjälpa honom att få ordning på denna. Detta kan ske genom att bl a försöka finna någon princip i uppbyggnaden, presentera delar av databasen och ge användaren möjlighet att editera i denna.

En intern databas skall naturligtvis också kunna lagras på externt medium. Det är också troligt, att man vill dela upp en större databas på flera fysiska filer t ex genom att särskilja olika substrukturer. Hur detta skall gå till har ännu inte diskuterats, men problemet är antagligen inte trivialt. Ett databashanteringssystem bör också innehålla rutiner för bearbetning av externa databasfiler. Det kan gälla att extrahera vissa strukturer eller merga ihop flera strukturer till en. Hela denna problematik och i synnerhet den yttre organisationen av databaser, pekar på att systemet även kräver en databasövervakare. Dess uppgift består i att

"veta" hur olika filer logiskt är relaterade till varandra, hur en uppdatering av en fil påverkar andra filer, vilka filer som är laddade för tillfället osv.

Framtiden kommer att ägnas åt att studera det relaterade problemområdet närmare. Inte bara ur PLAST:s egen, utan också ur mer allmän LISP- eller smådatabassynpunkt är ett sådant system intressant, och eventuella resultat av studier och experiment skulle kanske kunna appliceras på det vidare forskningsfältet små databaser. Det är heller inte otroligt, att idéer som uppstår ur den kommande verksamheten skulle kunna överföras även till andra programmeringsspråk med liknande möjligheter till datastrukturering, framför allt då SIMULA.

Referens

1. Olle Willén: Filhantering och procedureditering i PLAST för DEC-10. DLU 75/4.

4c. REC

REC är ett programmeringssystem som har en mycket enkel grundstruktur samtidigt som det är kraftfullt och användarvänligt. REC innehåller en del, helt eller delvis, nya idéer för programmeringsspråk. Speciellt bygger den på den variabelfria kalkylen (se ref 1). Detta betyder att inga variabelbindningar behöver göras vid funktionsanrop. Detta i sin tur för med sej att inga särskilda procedurdefinitioner behövs, ty varje label i programmet definierar en procedur. Inte heller behövs några goto's, ty ett goto motsvarar då ett proceduranrop utan parametrar och utan återhopp. Detta i sin tur gör att de program man skriver i REC automatiskt blir välstrukturerade, så att risken för programmeringsfel minskar.

Den idé som har tillkommit som ny under 1974 är följande: I LISP är program och data samma slags struktur. I REC däremot är båda program, data och kontrollstack samma slags struktur. Detta gör det möjligt för användaren att manipulera på kontrollstacken precis som han vill, och därigenom kan han införa nya former av program-kontroll i sitt program. Speciellt kan han införa back-tracking och parallella processer. Back-tracking används speciellt vid icke-deterministisk programmering. Parallella processer används speciellt vid simuleringar.

Denna nya idé har även medfört att den grundläggande filosofin bakom REC-systemet har ändrats. Tidigare kunde en funktion betraktas som en svart låda i vilken man stoppade ett argument, och därpå fick tillbaks ett värde från funktionen. Numera kan funktionerna betraktas som svarta lådor, som man stoppar in en tvådelad struktur i, ett argument och en kontroll-stack. Man får aldrig tillbaks något värde från funktionen, utan kontrollstacken innehåller all information om vart resultatet ska skickas.

Funktionerna kan liknas vid byråkrater vid ett statligt verk, som skickar papper mellan varandra. Framsidan av pappret innehåller det som ska bearbetas, medan baksidan av pappret innehåller uppgift på till vem det färdigbearbetade pappret ska skickas.

REC i MACRO-10

Under sommaren 74 gjordes ett REC-system i MACRO-10, maskinspråket på DEC-10 i Stockholm. Detta system är, trots att det är fullständigt, mycket litet när det gäller utrymme. Så är den maskinkodade delen av systemet endast 2 K ord (36-bits-ord), medan den REC-kodade delen är 6 K ord.

Men då är väl att märka, att symboltabellen upptar hela 3 K av dessa 6 K, beroende på att jag inte har brytt mej om att göra någon effektiv implementering av denna, utan i stället har haft en lätt modifierbar utformning på symboltabellen, ifall jag skulle vilja ändra utformningen av den. Men detta betyder i alla fall att REC är ett utmärkt språk för minidatorer, ty REC kombinerar ett litet system med en stor kraftfullhet.

I detta nya REC-system finns faciliteten med manipulerbar kontrollstack. Där kan man t ex plocka fram kontrollstacken, spara undan den, bygga upp en helt ny kontrollstack, och lägga dit den nya stacken. Tyvärr har jag funnit det vara ganska bökigt att manipulera på kontrollstacken i det nuvarande systemet, så under 1975 kommer jag att göra ett nytt REC-system i MACRO-10, som ska ha mycket smidigare metoder för stackmanipulation.

Backtracking

Med backtracking menas att man sparar tillståndet av en process i en viss punkt, för att senare gå tillbaka till denna punkt. När man kommer tillbaks ska alla ändringar vara ogjorda, precis som om processen aldrig hade lämnat denna punkt.

Med ickedeterministik programmering avses att man har någon slags SELECT-funktion, som helt slumpmässigt väljer något alternativ ur en mängd möjliga alternativ. Senare i programmet kan man "faila" om man inte är nöjd med det alternativ som valdes, och då återvänder man till selecten och väljer ett nytt alternativ. I praktiken gör man ofta så att man först väljer det första alternativet, sedan det andra osv. Om alla alternativen tar slut, gör man en ny fail och går tillbaka till den närmast föregående selecten. Detta betyder att man gör en slags djupet-först-sökning bland de möjliga alternativen.

Select-en här fungerar så att man sparar undan kontrollstacken i en lista. Sedan går man en väg ur selecten. Gör man senare en fail, så återställs stacken, och man är tillbaks i selecten igen. Denna gång går man en annan väg ut ur selecten, vilket innebär att man väljer ett nytt alternativ.

I REC ser select ut så här:

```
'select: if empty rest then x else
         savestack;
         if failflag then select rest else x !
```

Savestack är den funktion som sparar undan stacken, samtidigt som den slår av failflag. När man gör en fail slås failflag på igen och stacken återställs. Samtidigt återställs även alla reversibla tilldelningar. Det första if-et i select testar om endast ett alternativ återstår, och i detta fall går man bara rakt på.

Parallella processer

Parallella processer, eller CORUTINER som det även kallas, används speciellt vid simuleringar, ty där har man ju processer som ska försigå samtidigt.

Vid simuleringar har man först och främst en ACTIVATE- och en PASSIVATE-funktion. När ett objekt ska passivisera sej själv, i väntan på att någon annan process ska bli färdig och aktivera den igen, så lägger PASSIVATE ner kontrollstacken på objektet självt, varpå en ny kontroll-stack plockas fram från en global lista. ACTIVATE-funktionen fungerar så att kontrollstacken läggs ner på denna globala lista (så att det aktiverande objektet återfår kontrollen), varpå den nya kontrollstacken plockas fram från det objekt som ska aktiveras.

För att simulera att en process tar en viss tid, så har man en WAIT-funktion, som fungerar som PASSIVATE, bortsett från att i detta fall läggs kontrollstacken ner i en tidskö, i stället för på objektet självt. Man har då även en process som har som enda uppgift att aktivera objekten på tidskön.

Referens

1. Torgny Tholerus: The variable-free calculus or Lâmbda calculus without dummy variables, DLU 74/29.

2. Datalogilaboratoriet 1974, avsnitt 4g REC, sid 42-47.

4d QLISP

Introduction

The QLISP language was originally developed at the ARTIFICIAL INTELLIGENCE CENTER of Stanford Research Institute. (1) A version of this language is now available at the Datalogilaboratoriet and is implemented to run with INTERLISP/360 developed in Uppsala. Most of the features of the SRI-QLISP are available in this version which provides a variety of data-types, a data-base for associative storage and retrieval of expressions, a powerful pattern-matching capability, pattern-directed function invocation with an extended range of control structures and a mechanism for manipulating data contexts. These features are embedded in the programming environment of INTERLISP which provides a versatile, list-structure-oriented editor, an easy-to-use file package for maintaining symbolic files, and several debugging aids. A large number of the differences between this version

and the one running at SRI lie in the implementations of the pattern-matcher and the backtracking mechanism. Although the implementation of these features had in some cases to be adapted to the INTERLISP/360, this is transparent to the user. Programs written for the SRI-QLISP will run with only minor modifications to the source programs (mainly character-set conversion). Much effort has been dedicated to maintain compatibility with the SRI system.

The pattern matcher used in this version is based upon a generalised unification algorithm which unifies "bags" and "classes" as well as "tuples". This matcher is also available as a separate package (independent from the rest of QLISP). Furthermore since the "History" feature is not yet available in our INTERLISP, a temporary package has been put together to do backtracking. This is also used to do backtracking within the new pattern matcher but will be replaced by the regular History package as soon as this becomes available.

The main features of QLISP will be described briefly in the next pages to give a flavour of its capabilities. The reader is referred to the user manual (1) for a more detailed description of the language and its syntax.

QLISP Expressions

QLISP provides complete freedom in intermingling LISP expressions with QLISP statements that provide net storage and retrieval, pattern matching, and control structure manipulation. As an example, consider the ARE-COUSINS program:

```
(QLAMBDA (←PERSON1 ←PERSON2)
         (IS (FATHER $PERSON1 ←F))
         (IS (UNCLES $PERSON2 ←←U))
         (IF (MEMBER $F $U)
             THEN (PRINT (&($PERSON1 AND $PERSON2 ARE COUSINS)))
             ELSE (PRINT (&($PERSON1 AND $PERSON2 ARE NOT COUSINS]
```

See the QLISP user manual (1) for a more precise description of the syntax of QLISP expressions; for the time being, let us only say that the characters $, ←, ←← are variable prefixes. Roughly speaking, when we want to use a

previously assigned value of a variable, we give it a $ prefix; when we want to assign a variable a new value we give it a ← prefix. Double prefixes designate fragment variables which may stand for more than one element. When this program is executed, two associative retrievals from the discrimination net will obtain the father of the first person and the uncles of the second person. If the father of the first person is among the uncles of the second, we proclaim the two persons to be cousins.

QLISP Data Structures

QLISP provides a rich set of data structures and primitives for manipulating them. In addition to all the data types available from INTERLISP, QLISP provides four data types to represent sets: TUPLE, VECTOR, BAG and CLASS. Tuples and vectors are ordered sets (thus equivalent to a LISP list) and differ only when they are evaluated. The value of a tuple is the result of applying its first element to the argument list represented by the values of the rest of the elements. The value of a vector is a vector of the values of its elements. Thus, a tuple is useful for representing a form containing a function and its arguments, whereas a vector will be used for representing an argument list.

Bags and classes are unordered collections of elements. Elements in a bag may be duplicated. For example, (BAG A A B C) is equivalent to (BAG A C B A), but is different from (BAG A B C). No duplication is allowed for elements in a class. For example, (CLASS A A B C) is equivalent to (CLASS C B A).

QLISP provides also data of type IDENTIFIER and NUMBER which are equivalent to LISP's literal atoms and numbers but are uniquely represented in the QLISP data base (see below).

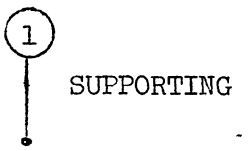
The QLISP Data Base, a Discrimination Net

QLISP provides a variety of features for building, updating and accessing a data base. Expressions composed of any of the data types described above may be placed in the data base. Apart from identifiers and numbers, the data base is maintained in the form of a discrimination net. The net is a tree-like structure, in which the nodes represent tests to apply to an expression and the branches represent the values returned by the

tests. In general, these tests are set up to find the first difference, scanning left to right, between two expressions.

Although an understanding of the internal structure of the discrimination net is unnecessary for the casual user, it will aid planning of efficient representations. Here is an example of how the discrimination net is updated while we are building a model of a robot about to throw a wad of paper into a wastebasket.

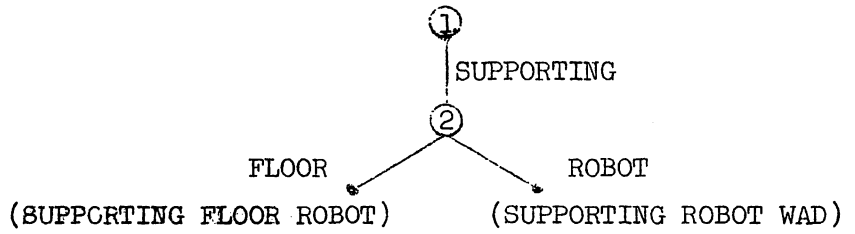
The first assertion we will make is (SUPPORTING FLOOR ROBOT). At this point, the net may look like this:



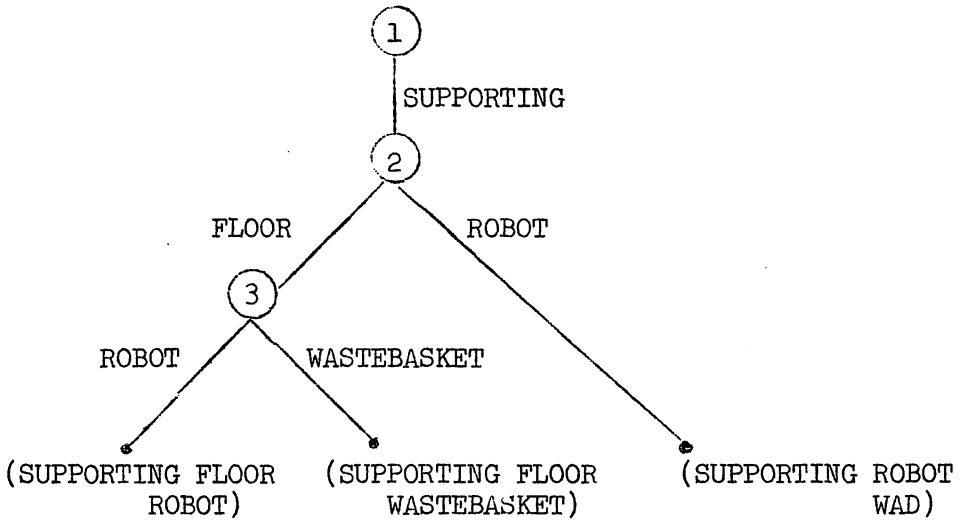
(SUPPORTING FLOOR ROBOT)

The node with a 1 in it represents a test to be made on the first element of an expression. The line labelled SUPPORTING indicates that all expressions whose first element is SUPPORTING will be found below it. The solid node represents a terminal node of the net, where the expressions themselves are stored. This terminal node contains the tuple (SUPPORTING FLOOR ROBOT).

Let us now assert (SUPPORTING ROBOT WAD). The data base updating mechanism first checks to see if the expression is already in the net. The top node in the net says to discriminate on the first element of the expression. The first element is SUPPORTING, so we follow the branch labelled SUPPORTING. This brings us to a terminal node, so we check to see if the expression in the net is the same as the expression we are adding. In this case, they are not equal, so the terminal node must be transformed into a testing node. The two expressions are scanned left to right for the first difference, in this case in the second position. The net is updated to reflect this new discrimination, and it now looks like this:



If we now assert (SUPPORTING FLOOR WASTEBASKET), a similar transformation of a terminal node into a testing node will make the net look like this:



Canonical Representation of Expressions

By storing all data in a common discrimination net, QLISP can represent equivalent expressions uniquely. In the QLISP net, only one instance of an expression may occur. Before an expression is entered into the net, it is transformed into a canonical form. A new terminal node will not be created if the expression already exists in the net. Thus, for example,

(BAG A B C) and (BAG B C A)

are not only equivalent, they are exactly the same pointer into the discrimination net.

Thus, arbitrary expressions are represented uniquely in QLISP, just as atoms are in LISP. Therefore it is possible to assign properties to QLISP expressions in the same way as to LISP atoms. For instance, we may execute the command:

(QPUT (PLUS A B (MINUS A)) SIMPLIFIESTO B),

which will put the value B under indicator SIMPLIFIESTO in the property-list of the expression (PLUS A B (MINUS A)). If we ever encounter this expression, or any equivalent expression, we can look on its property-list and find a simplification for it.

The canonical representation of expressions in the net has an important implication for programming in QLISP. References to net-expressions are not merely EQUAL; they are EQ.

The Pattern Matcher

Expressions are taken apart and their components are named through the use of the pattern language. Pattern matching takes place during associative retrievals from the data base, the binding of QLAMBDA patterns (pattern directed function invocation) and explicit calls to the matcher.

Patterns and arguments can be arbitrary QLISP expressions. A simple example is this explicit call:

```
(MATCH (VECTOR ←X ←Y) (VECTOR 1 2))
```

The pattern (VECTOR ←X ←Y) is matched with (VECTOR 1 2) and variables X and Y will be bound to 1 and 2 respectively. If the pattern matcher cannot match a pattern with an argument, a condition known as failure will occur. (see "backtracking" below) The same decomposition process takes place during QLAMBDA binding.

Example: ((QLAMBDA (TUPLE ←X ←Y) (TUPLE $Y $X)) (TUPLE 1 2))
will return (TUPLE 2 1) as its value.

IS is a QLISP statement to retrieve expressions from the data base.

Example: (IS (RED ←OBJECT)) searches the data base for an object asserted to be RED and binds OBJECT to that object.

As an example of the gain in clarity which a pattern matching facility provides, consider the following example: Suppose L is a list of the form (X Y (V W)) and we want to set variable K to the rearranged list (V W (X Y)). In LISP we would write:

```
(SETQ M (CADDR L))
(SETQ K
      (APPEND M
              (LIST (LIST (CAR L) (CADR L)))))
```

Although the operation described here is conceptually simple, it is quite impossible to see what is going on. In QLISP we would write:

```
(MATCHQQ (←X ←Y (←V ←W)) $L)
(MATCHQQ ←K ($V $W ($X $Y)))
```

The QLISP representation is clearer because it is more pictorial. By admitting patterns with classes, bags, or fragments, we have introduced an element of nondeterminism: the possibility that a pattern may match the same expression in more than one way.

Example: (MATCH (CLASS ←X ←Y) (CLASS 1 2))

The two possible assignments are X=1, Y=2 and X=2, Y=1. The QLISP pattern matcher is able to recognize the different alternatives, to choose one, and to produce the "next" possible set of bindings on request.

Backtracking

The side-effects of QLISP computations may be undone (in the INTERLISP sense) by the use of the QLISP failure mechanism. A statement which invokes pattern matching will fail if no match exists or if all matches have been exhausted. Other statements may be caused to fail by the use of the FAIL statement.

A failure will cause a return to some backtrack point and undo all undoable computations performed since the backtrack point was established.

Manipulation of expressions in the net is undoable by default, but it can be controlled by the context mechanism, (see below). Manipulation of LISP data is undoable if it is done by means of "/ functions", provided by INTERLISP.

Backtrack points are established within all net storage and retrieval statements and within QLAMBDA expressions that have the BACKTRACK option.

Failure may be caused explicitly by executing the FAIL statement. Its format is (FAIL name).

If name is absent or NIL, FAIL causes a failure.

If name is CALLER, FAIL causes the last net storage or retrieval statement to fail.

If name matches the NAME of a net storage or retrieval statement, FAIL causes the named statement to fail.

Pattern directed function invocation

QLISP functions are of three varieties: LAMBDA and NLAMBDA, as in INTERLISP, and QLAMBDA. A QLAMBDA expression is of the form

(QLAMBDA bv e_1 {e_2 ... e_n} {BACKTRACK}).

The bound variabel part, bv, is a pattern which is to be matched against the argument of the QLAMBDA expression. QLAMBDA functions are defined and applied in a similar manner to LAMBDA functions. Since the bound variable part bv is a single pattern, the match which takes place when the QLAMBDA function is entered can only succeed if there is a single argument. So if multiple arguments are given, they are grouped into a tuple at the time the function is applied. For example suppose the function REV has definition

```
(QLAMBDA (←X ←Y)
         (&($Y $X)))
```

This is a function that reverses a list of length 2. (←X ←Y) is the bv and (&($Y $X)) is e_1. When (REV(QUOTE(1 2))) is evaluated, the pattern (←X ←Y) is matched against the argument (1 2). Then REV returns the tuple (2 1). When (REV 1 2) is evaluated, the arguments are grouped into a tuple (1 2) and evaluation of the function proceeds as before.

The main use of the pattern directed function invocation however, is in connection with the QLISP data storage and retrieval statements and the GOAL statement whereby the programmer does not specifically call functions for doing a job but merely specifies the task he wants to be executed. The appropriate procedures, (those whose pattern matches the goal) will then be invoked.

Since nondeterministic matches may take place, there may be more than one possible binding of the variables in the bound variable part. For example,

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

applied to (BAG 1 2) could evaluate to either (1 2) or (2 1). For such applications QLISP does not normally establish a backtracking point and performs the alternate bindings and evaluations on failure. It uses the first binding chosen by the pattern matcher. Since it does not establish a backtracking point, a failure bypasses this potential choice point. If the user wishes alternate bindings to be invoked on failure, the BACKTRACK option must be specified.

The Context Mechanism

All properties associated with an expression in the QLISP net can be stored and retrieved with respect to a "context," a scope for binding a variable or, more generally, a scope for assignment of properties to an expression. Under the same indicator, one expression can have different properties with respect to different contexts.

The QLISP context mechanism provides scope restrictions for values and properties of expressions. These restrictions are created by the block and calling structure of QLISP programs in a manner similar to that for values of variables in INTERLISP. In addition, QLISP provides the means for creating scopes independently of these structures. New contexts may be created and old contexts pushed or popped using the CONTEXT statement. Almost all the QLISP statements provide for a context option thus pattern matching and associative storage and retrieval may occur with respect to a specified context.

To make an assignment of a value to a variable or of a property to an expression with respect to a context means that the new value or property is available within the context, but that any old value or properties are still retained outside that context.

DLU-QLISP

With QLISP implemented at the Datalogilaboratoriet, we have the opportunity to run and study most of the ideas introduced by the recent generation of AI-languages. QLISP promises to become one of the most useful and is in fact already in use by several AI communities to do problem solving,

planning, automatic programming, vision etc ... We think its success is due mainly to the fact that QLISP is implemented as a collection of INTERLISP packages rather than as an interpreted language on top of INTERLISP (as many of its predecessors are). Several QLISP features seem to be relevant to other work going on at DLU, so some work will be done to make independent packages for these features and incorporate them with the common LISP program library. The pattern-matcher is already available as a separate package. No further development of the language is planned at DLU. As a separate project however, we would like to improve the discrimination net. Some aspects and ideas for this work are suggested in Section 17.

References

1. Reboh, René and Sacerdoti, Earl. A Preliminary QLISP Manual. SRI AI Center Technical Note 81, August 1973.

2. Bobrow, Daniel G. and Raphael, Bertram. New Programming Languages for AI Research. SRI AI Center Technical Note 82, August 1973.

3. DATALOGILABORATORIET. Årsrapport 1974, sektion 16. AI-språk.

4e. SLISP - Simulering med LISP som programmeringsspråk (Ref. DLU 74/35)

Bakgrund:

Under hösten 1974 fick DLU i uppdrag av FOA att utföra en pilotstudie av en simuleringsmodell med LISP som programmeringsspråk. FOA var framförallt intresserade av frågorna:

- Vilka fördelar kan erhållas genom att utnyttja tekniken för små databaser.
- Hur skall interaktionen mellan människa(operatör) och simuleringsprogram utföras.

En viktig skillnad mellan dessa punkter är att i det förra fallet har vi valt LISP som programmeringsspråk, och de fördelar detta för med sig undersöks. I det senare fallet ville man experimentera sig fram till ett kommandospråk som skulle kunna användas även av andra simuleringssystem.

Uppdraget har delats i två klart avskiljbara delprojekt, dels ett programpaket för simulering medelst LISP (SLISP), dels har paketet använts för att skriva program för den av FOA specifiserade modellen. I detta kapitel (4e) beskrivs SLISP. I kapitel 13e redogörs för en simuleringsmodell med hjälp av SLISP.

Kort karakteristik av SLISP

Termer

objekt - båtar, bilar, bussar etc (de föremål som gör någonting i simuleringen).

klass - sort (typ) av objekt.

kö - samling av objekt (first in first out)

tidskö - SLISP-systemets kö av inplanerade händelser (i tidsordning)

varje objekt har:

statiska attribut

(gemmansamma för alla objekt i klassen)

dynamiska attribut

(unika för varje objekt)

tillhörande kod, som beskriva objektets "roll" i simuleringen.

varje objekt kan bl a

- aktiveras dvs placeras först i tidskön och börja exekvera tillhörande kod, där det förut stannade.
- sättas in och tas bort från köer.
- aktivera andra objekt.
- stanna sig själv, antingen en viss tid, eller tills något annat objekt startar sig.

Jämförelser med SIMULA

Allm. SLISP har utformats så att program skrivna i SLISP skall vara enkla att överföra till SIMULA. Så har t ex de flesta SIMULA-primitiver en analog motsvarighet i SLISP (hold, activate etc). SLISP själv är utformat ungefär som ett runtime-system för SIMULA (med "tidskö" för

inplanerade händelser) vilket gör att logiken i ett SIMULA-program och ett SLISP-program blir densamma.

Några skillnader

Inskränkningar:

- I SLISP kan klasser ej konkateneras.
- Tillsvidare är detach och resume ej direkt åtkomliga för programmeraren (Observera dock, att om activate, hold etc används skall man ej heller i SIMULA använda detach eller resume.)
- Slumptals generering är ej implementerat.
- I/O annorlunda. Sträng-hanteringsprocedurer finns ej (förutom sådana som finns i det använda LISP-systemet).
- Pga den tekniska implementeringen måste varje anrop som föranleder ett temporärt avbrott i ett objekt (dvs utför detach på något sätt) dels ange namnet på en procedur där exekveringen senare skall återtagas, dels skall anropet avsluta proceduren. Rent praktiskt blir man alltså i SLISP tvungen att stycka upp ett program-avsnitt i flera procedurer, där man i SIMULA hade klarat sig med en enda procedur.

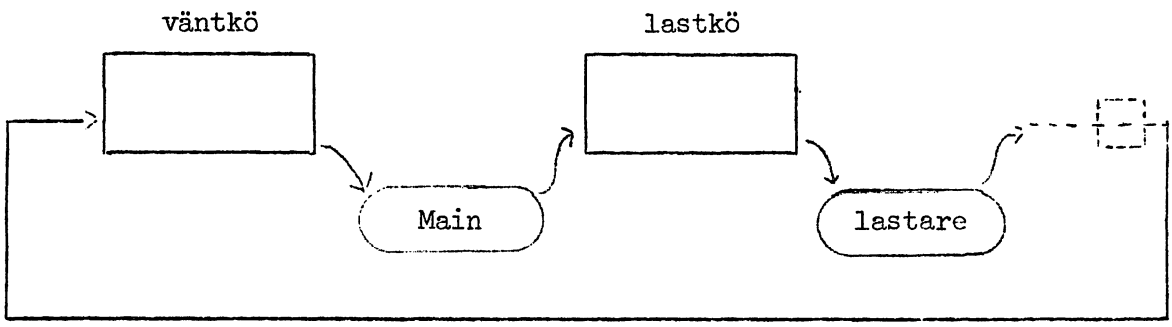
Utvidgningar:

- Begreppet "Static attribute". Attribut associerat till en klass. Attributets värde gäller för alla medlemmar till klassen.
- debug-faciliteter.
  Här erhålles stora fördelar, framförallt vid interaktiva körningar. Förutom LISP's egna debug-faciliteter (edit, trace etc) finns följande:
  - SIMPRINT. Ger snygga utskrifter av köer, objekt etc.
  - Möjlighet att sätta brytpunkter (där t ex interagering skall ske)
  - Möjlighet till egna feltester på ett enkelt, standardiserat sätt.
- Begreppet "räknade objekt". Efter anropet (COUNT LASTBIL) är varje enskilt objekt ur klassen LASTBIL åtkomligt via (LASTBIL nr). Även utskrifter från SIMPRINT har detta format.

- Ett kommandospråk att användas vid interagering med programmet. Detta är ännu ej fixerat, och redogörs för närmare i kapitel 13e. (Man kan naturligtvis även interagera direkt via LISP)

Ett exempel

Följande exempel är avsett att illustrera olika faciliteter i SLISP. Kommandospråket exemplifieras dock i kapitel 13e.



Modellen beskriver lastbilar som börjar i en väntkö. En process main (här huvudprogrammet) tar bilar från väntkön och placerar dem i lastkö. En process lastare tar bilar från lastkö, lastar dem (vilket tar en viss tid) och skickar dem vidare.

För att förenkla modellen åker bilarna sedan tillbaka till väntkön (vilket också tar en viss tid).

Varje bil har attributen åktid och lasttid.

Definition av klassen bilar

SLISP

```
(DEFOBJ BIL (LASTTID ÅKTID)
        BIL-START)


(DE BIL-START()
    (INTO VÄNTKÖ (THIS))
    (PASSIVATE BIL-A>
(DE BIL-A()
    (INTO LASTKÖ (THIS))
    (COND ((IDLE LASTARE)
            (ACTIVATE BIL-B>
                  LASTARE))
          (T (BIL-B>
(DE BIL-B() (PASSIVATE BIL-C
(DE BIL-C()
    (HOLD BIL-D ÅKTID>
(DE BIL-D()
    (WAIT BIL-A VÄNTKÖ>
```

SIMULA

```
process class bil(lasttid, åktid);
   real lasttid, åktid;
   begin


      into(väntkö);
      passivate;
bila:
    into(lastkö);
    if lastare.idle then
       active lastare;


passivate;

hold(åktid);

wait(väntkö);
goto bila
end;
```

Definition av klassen lastning

SLISP

```
(DEFOBJ LASTNING( X )
        LAST-LOOP>

(DE LAST-LOOP()
    (COND((EMPTY LASTKÖ)
          (PASSIVATE LAST-A))
          (T (LAST-A>
(DE LAST-A()
    (SETQ X (FIRSTQ LASTKÖ))
    (OUT X)
    (HOLD LAST-B
       (GETDYNATTR X LASTTID>

(DE LAST-B()
    (ACTIVATE LAST-LOOP X>
```

SIMULA

```
process class lastning( X );
   ref(bil) X;
   begin
   lastloop:
      if lastkö.empty then
         passivate;

   X:- lastkö. first;
   X.out,

   hold(X.lasttid);

   activate X;
   goto lastloop
   end;
```

Definition av huvudprogrammet

SLISP

```
(DE BEGIN()


(PROG(LASTKÖ VÄNTKÖ
    X LASTARE)

(SETQ LASTARE
    (NEW LASTNING NIL))
(SETQ VÄNTKÖ
    (NEWHEAD VÄNTKÖ))
(SETQ LASTKÖ
    (NEWHEAD LASTKÖ))

LOOP
(PRINT 'MAIN-PROGRAM)
(SIMPRINT (EVAL (READ)))
(GO LOOP>
```

SIMULA

```
SIMULATION begin
   process class bil ...
   process class lastning ...
   ref(head) lastkö, väntkö;
   ref(lastning) lastare;

   lastare :- new lastning(none)

   väntkö :- new head;

   lastkö :- new head;

   LOOP:
   anrop av interagerande
   procedur
      goto loop
   end;
```

En körning: (input är det som följer omedelbart efter en :*)

```
:*(SIMULATION BEGIN)

"MAIN-PROGRAM" :*(ACTIVATE BEGIN (NEW BIL 1 3))
NIL

"MAIN-PROGRAM" :*(ACTIVATE BEGIN (NEW BIL 5 6))
NIL

"MAIN-PROGRAM" :*(ACTIVATE BEGIN(NEW BIL 11 12))
NIL

"MAIN-PROGRAM" :*(ACTIVATE BEGIN (FIRSTQ VANTKO))
NIL

"MAIN-PROGRAM" :*(TIMEQUEUE)
THE SYSTEMS TIME QUEUE =
       0    MAINPROGRAM
       1    LASTNING

"MAIN-PROGRAM" :*VANTKO
QUEUE VANTKO
      MEMBERS =
      (BIL 2)
      (BIL 3)

"MAIN-PROGRAM" :*(FIRSTQ VANTKO)
OBJECT (BIL 2)
      DYNAMIC ATTRIBUTES =
      LASTTID - 5
      AKTID - 6
      DETACHED AT BIL -A
      MEMBER IN QUEUE VANTKO

"MAIN-PROGRAM" :*(BIL 3)
OBJECT (BIL 3)
       DYNAMIC ATTRIBUTES =
       LASTTID - 11
       AKTID - 12
       DETACHED AT BIL -A
       MEMBER IN QUEUE VANTKO

"MAIN-PROGRAM" :*(HOLD BEGIN 2)
NIL

"MAIN-PROGRAM" :*(ACTIVATE BEGIN (FIRSTQ VANTKO))
NIL

"MAIN-PROGRAM" :*(ACTIVATE BEGIN (FIRSTQ VANTKO))
NIL
```

3 bilar aktiveras och ställer sig i VANTKO.

1:a bilen i VANTKO aktiveras

Titta på tidskön. 1:a bilen har börjat lassa

Titta på VANTKO och LASTKO

Titta på bil nr 3

Vänta 2 och aktivera sedan

de 2 andra bilarna

```
"MAIN-PROGRAM" :(TIMEQUEUE)
THE SYSTEMS TIME QUEUE =
      2     MAINPROGRAM
      4     (BIL 1)
      7     LASTNING

"MAIN-PROGRAM" :VANTKO
QUEUE VANTKO
      EMPTY

"MAIN-PROGRAM" :LASTKO
QUEUE LASTKO
      MEMBERS =
      (BIL 3)
```

Titta på köerna.
Bil 1 åker.
Bil 2 lastas (LASTNING aktiv)
Bil 3 väntar på lastning.

```
"MAIN-PROGRAM" :(HOLD BEGIN 3)
NIL
```

vänta 3

```
"MAIN-PROGRAM" :(BIL 3)
OBJECT (BIL 3)
       DYNAMIC ATTRIBUTES =
       LASTTID - 11
       AKTID - 12
       DETACHED AT BIL -C
       MEMBER IN QUEUE LASTKO
```

Titta på bil nr 3

Bil nr 3 skall så småningom försätta

```
"MAIN-PROGRAM" :(PUSHBREAK BIL -C (SIMERR "PUNKTEN BIL -C ANROPAD"))
BIL -C

"MAIN-PROGRAM" :(BREAKON BIL -C)
(BIL-C)
```

Lägg in en break i punkten BIL-C

```
"MAIN-PROGRAM" :(HOLD BEGIN 10)
```

vänta 10

```
"SIMERR CALLED"
"PUNKTEN BIL-C ANROPAD
"SIMERR - LOOP" :(TIMEQUEUE)
THE SYSTEMS TIME QUEUE =
      7      (BIL 2)
      7      LASTNING
      15     MAINPROGRAM
```

Mitt break-meddelande.
Jag ligger nu i breakpunkten och interagerar

```
"SIMERR - LOOP" (THIS)
OBJEKT (BIL 2)
       DYNAMIC ATTRIBUTES =
       LASTTID - 5
       AKTID - 6
       EXECUTING IN BIL-C
```

Titta efter, vilken bil som passerar BIL-C. Svar nr 2

```
"SIMERR - LOOP" :CONTINUE
```

Fortsätt simuleringen

```
"MAIN-PROGRAM" :VANTKO
QUEUE VANTKO
      MEMBERS =
      (BIL 1)
```

Bil 1 har gått runt.

```
"MAIN-PROGRAM" :(HOLD BEGIN 10)

"SIMERR CALLED"
"PUNKTEN BIL-C ANROPAD"

"SIMERR - LOOP" :(THIS)
OBJECT (BIL 3)
       DYNAMIC ATTRIBUTES =
       LASTTID - 11
       AKTID - 12
       EXECUTING IN BIL-C

"SIMERR - LOOP" :CONTINUE

"MAIN-PROGRAM" :VANTKO
QUEUE VANTKO
      MEMBERS =
      (BIL 1)
      (BIL +)

"MAIN-PROGRAM" :(TIMEQUEUE)
THE SYSTEMS TIME QUEUE =
      25    MAINPROGRAM
      32    (BIL 3)
```

Vänta 10

En till bil passerar BIL-C. Vilken?

Nr 3.

Även bil nr 2 har gått runt.

Bil 3 på väg hem...
osv.

5. TILLFÖRLITLIGHET I PROGRAMMERINGSSYSTEM

Under hösten 1973 utfördes vid DLU arbete inom ett projekt "totala programmeringssystem" där målsättningen var att specificera de funktioner och understödsprogram som skall finnas i ett totalt programmeringssystem för ett konventionellt programmeringsspråk. Framför allt studerades inkörnings- och testhjälpmedel, och målet var att det specificerade systemet skulle möjliggöra utveckling av program med väsentligt större tillförlitlighet än normalt. Det framstod dock klart att sådana hjälpmedel inte är tillräckliga för att man i konventionella programmeringsspråk på konventionella maskiner skall kunna utveckla program med tillräckligt stor tillförlitlighet.

Under våren 1974 ändrades därför projektets målsättning till ett grundläggande studium av programvarutillförlitlighet. Projektets långsiktiga mål är att specificera de ändringar som krävs i såväl programutvecklingsmetodik och system programvara som i maskinarkitektur för att tillförlitligheten i programvarusystem skall kunna höjas avsevärt över dagens nivå. Projektet är för närvarande inriktat på metoder för programutveckling eftersom det framstår som helt klart att programmeringsspråk och maskinarkitektur måste anpassas till programmeringsmetoderna och inte tvärtom.

Grundläggande begrepp

I detta avsnitt förklaras och definieras ett antal grundläggande termer och begrepp.

Två besvärliga faser av programutvecklingen är testningen av programmet och felsökningen i programmet. Med testning av ett program (program testing) menas den aktivitet då man försöker förvissa sig om att programmet gör vad det skall göra. Felsökning och rättning (debugging) av ett program tar vid först då man vid testningen funnit att programmet i något avseende inte gör vad det borde och innebär att felet lokaliseras och rättas.

Fel, som uppträder vid exekveringen av ett program, kan ha uppstått av flera orsaker och kan indelas i grupper efter orsaken. Ett fel kan definieras som vad som helst som gör att resultatet av programmets exekvering inte blir den det borde bli. Detta kan till exemple bero på brand,

strömavbrott, sabotage och liknande, eller på felaktiga indata, och felaktiga operatörsingrepp, men de fel man oftast diskuterar i tillförlitlighetssammanhang är programvarufel (software errors) och maskinvarufel (hardware errors). Programvarufel kan definieras som alla fel som introduceras i programmet under programutvecklingen, översättningen till objektkod etc fram till dess att programmet ligger som maskinkod klart att exekvera i målmaskinen. Hårdvarufelen är de fel hårdvaran gör vid interpreteringen (tolkningen) av maskinprogrammet.

Om ett program betraktas som en avbildning P från indata i till utdata u, där indata och utdata tillhör några mängder av tillåtna data och tillstånd, så är programvarufel något som gör att P är skild från P_c, där P_c är det fiktiva korrekta program som överensstämmer med intentionen för P. Ett maskinvarufel är då något som ändrar P så att P(i) är skilt från $P_c(i)$.

Programvarufel är alltså fel som alla finns i P från början och är kvar tills P ändras. Att man med programtestning inte kan finna alla fel med en gång beror på att man för detta måste jämföra P(i) med $P_c(i)$ för alla möjliga i. Detta beror på att ett programvarufel oftas gör att $P(i) \neq P_c(i)$ bara för en mycket liten delmängd av alla möjliga i och på att vi inte vet vilken denna delmängd är.

I alla sammanhang då man skall avgöra huruvida ett program är korrekt eller inte måste man ha något att jämföra med, dvs något fiktivt program P_c. Man kan inte välja P_c som det program som uppfyller specifikationerna för P eftersom en stor del av programvarufelen just är specifikationsfel. P_c bör vara det program som beställaren tänkt sig, men problemet att bestämma P_c har också anknytning till människa-maskin diskussionen och andra diskussioner om datorns roll i samhället. Ett program som är människovänligare än vad beställaren tänkt sig bör ju inte betraktas som felaktigt.

Programutveckling

Ett program kan beskrivas på följande utvecklingsnivåer:

- Intentionen

2. Specifikationen
3. Algorithmen
4. Programmet
5. Maskinkoden

Med intentionen menas specifikationen av det fiktiva "önskeprogrammet", dvs den fullständigt korrekta specifikationen av det önskade programmet. Specifikationen beskriver vad programmet skall göra, vilka in- och utdata det skall behandla etc. Algoritmen talar om hur programmet skall utföra det som beskrivs i specifikationen. Programmet är en formalisering av algoritmen skriven i något maskinläsbart språk och maskinkoden är den exekverbara formen av programmet som den genererats av tillämpliga program som kompilator, laddningsprogram etc.

Utvecklingen av ett program kan beskrivas som en serie övergångar mellan de ovan uppräknade nivåerna. Övergångarna mellan 1 och 2, 2 och 3 och mellan 3 och 4 utförs av programmerare eller systemmän medan övergången mellan 4 och 5 oftast görs av understödsprogramvara. Vid varje övergång kan också fel införas i programmet, fel som sedan fortplanteras genom följande övergångar.

Det är naturligtvis bara utvecklingen av mycket små och enkla program som kan beskrivas som fyra övergångar fram till ett färdigt maskinkodsprogram. För att beskriva utvecklingen av mer komplicerade program behövs kännedom dels om programmets uppbyggnad och dels om hur programmet byggts upp, dvs programmerings metodiken. Detta är nödvändigt eftersom ett program kan ses som ett stort antal skilda enheter som utvecklats ur varandra eller byggts upp med hjälp av varandra.

Programstruktur

Ett program kan beskrivas som en enhet, en svart låda, med klara specifikationer för indata, utdata och funktion. Programmet bör också ses som en instans av en algoritm bestående av kontrollstrukturer och mindre enheter, också dessa med specifikationer för indata, utdata och funktion. Algoritmen kan ofta vara mycket enkel, t ex bara en sekvens av enheter. Varje enhet i algoritmen kan sedan på samma sätt ses som en instans av en algoitm bestående av kontrollstrukturer och enheter.

Detta kan fortsättas tills enheterna är satser i det använda programmeringsspråket.

Om programmets struktur beskrivs på detta sätt måste två villkor vara uppfyllda för att programmet skall vara korrekt i förhållande till sin specifikation. Dels måste alla in- och utdataspecifikationer vara inbördes konsistenta, dvs utdata från en enhet måste stämma överens med indata till nästa enhet. Dessutom skall varje algoritm instans utföra den specificerade funktionen med hjälp av de ingående enheterna. Detta kan delas upp i två problem. För det första skall algoritmen visas utföra den specificerade funktionen. Att visa detta är ofta ett problem inom tillämpningsområdet för algoritmen, t ex numerisk analys för algoritmer för numerisk lösning av differentialekvationer etc. För det andra måste det visas att just den instans av algoritmen som ingår i enheten överensstämmer med algoritmen som avsågs. Detta är ett problem för systemutvecklaren eller programmeraren.

För att kunna visa att specifikationerna för de enheter som är satser i programmeringsspråk stämmer överens med vad satserna verkligen utför krävs att programmeringsspråket och den underliggande maskinvaran är väldefinierade (t ex axiomatiskt beskrivna) och att det t ex inte finns okontrollerade undantag från den normala funktionen som är så vanligt hos existerande system.

Metoder för programutveckling

Följande problem är väsentliga vid utvecklingen av ett program:

1. Framtagningen av specifikationen för programmet som helhet, dvs bestämning av indata, utdata och funktion hos programmet.
2. Framtagning eller konstruktion av algoritmer för att utföra specificerade funktioner med hjälp av underenheter. I detta ingår att specificera eller utnyttja specifikationerna hos underenheterna.
3. Översättning av algoritmerna till programmeringsspråk.

Detta är nästan samma uppställning som den som beskrev programutveckling

som en serie övergångar mellan programbeskrivningsnivåer i ett tidigare avsnitt. Det arbete som beskrivs i punkt 2 måste göras för varje enhet som ingår i programmet. Om en ren top-down-metod används vid programutvecklingen innebär detta att för varje enhet som specificerats tas en algoritm fram och de i denna ingående underenheterna specificeras. Dessa behandlas sedan på samma sätt tills samtliga enheter består av satser i programmeringsspråket.

Om i stället en bottom-up-metod används byggs enheterna upp av tidigare specificerade enheter. Algoritmerna måste då väljas så att de kan utnyttja egenskaperna hos dessa tidigare specificerade enheter. Motsvarande problem vid top-down-utveckling är att varje uppdelning i underenheter måste göras så att den så småningom kan leda fram till satser i programmeringsspråket. Detta kan sägas vara det fjärde stora problemet vid programutveckling, dvs att uppdelningen eller uppbyggnaden måste syras mot ett i början ganska avlägset mål. För top-down-metoden är detta mål programmeringsspråket och målmaskinen och för bottom-up-metod är det det färdiga programmets specifikationer.

I praktiken används sällan någon av dessa metoder i sin rena form. Program utvecklas ofta samtidigt uppifrån och nedifrån eller till och med samtidigt från flera nivåer och utåt tills nivåerna möts. Vilken programmeringsmetodik man än använder kommer det färdiga programmet att bestå av ett antal enheter. Mellan dessa enheter råder olika relationer. En självklar sådan relation är "ingår i algoritmer för" som anger hur programmet är strukturerat, men även relationer som visar hur programmet är utvecklat, t ex "är använd vid utvecklingen av", är väsentliga. Relationerna måste vara kända för att ändringar i programmet och felsökning och rättning skall kunna utföras utan att nya fel införs. Relationerna måste också vara kända under programutvecklingen eftersom varje programutvecklingsmetod måste bygga på att specifikationen inte är statisk utan kommer att modifieras både under utvecklingsfasen och senare.

Den information som måste finnas tillgänglig under och efter utvecklingen av ett program är alltså specifikationerna för varje enhet, algoritmen

för varje enhet och relationerna mellan enheterna. Varje bra programmeringsmetodik måste möjliggöra att denna information sparas och finns tillgänglig. Skall dessutom automatiska metoder för t ex kontroll av överensstämmelser mellan specifikationer användas måste informationen formaliseras så att den blir maskinläsbar.

Projektinriktning

De ovan beskrivna principerna utgör några av utgångspunkterna för arbetet inom projektet. Projektet kommer att fortsätta dels med vidare arbete på programutvecklingsmetoder och dels med mindre detaljstudier av närliggande områden. Arbetet på programutvecklingsmetoder är det mest väsentliga eftersom alla åtgärder för att väsentligt höja programvarutillförlitligheten måste ha sin grund i metoder för programutveckling och dessa metoders egenskaper.

En redan genomförd studie av ett mindre område behandlade klassificering av programvarufel. Denna finns dokumenterad i DLU 75/1 (Software Error Classification, 1975-01-11).

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:

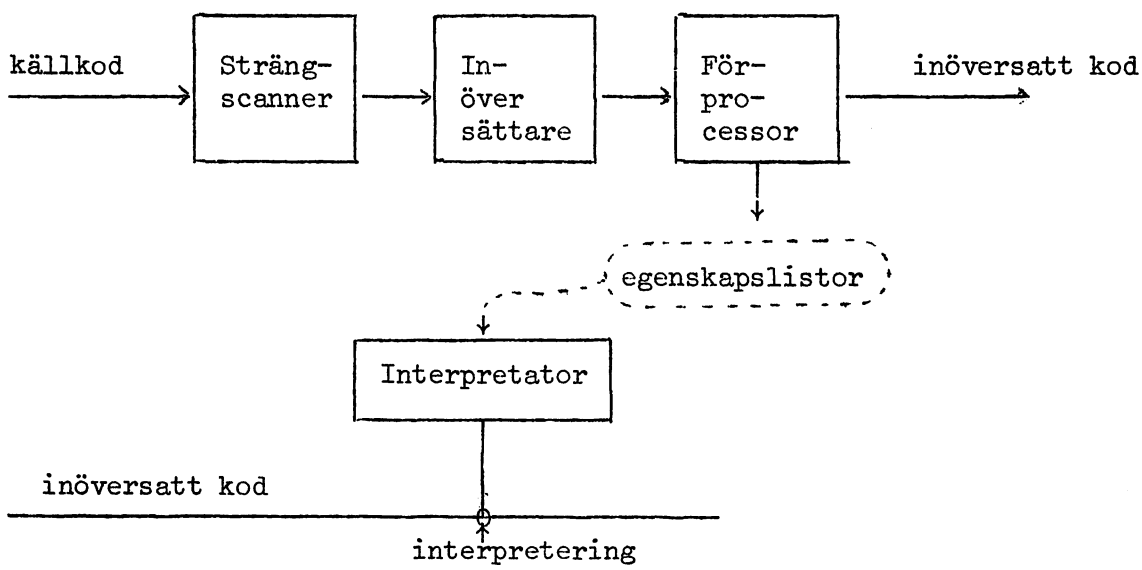
- att ge en formell beskrivning av SIMULA:s semantik
- att, med en formella beskrivningen, kunna exekvera SIMULA-program

Som värdefulla biresultat kan nämnas:

- ett interaktivt SIMULA-system (om man har ett interaktivt LISP-system)
- Debugging-faciliteter under exekveringer, som t ex gå in och titta på variabelvärden, editera programmet, titta hur en anrops-sekvens ser ut osv (jfr även SLISP, kap 4e)
- en definition av SIMULA, som även talar om vad som sker under kompilering och vad som sker under exekvering
- en typechecknings rutin, som kontrollerar att programmets hantering av olika datatyper är korrekt. Kontrollen utföres före interpreteringen och kan användas för att ta bort onödiga run-time-kontroller eller för att flagga för fel som annars skulle inträffa under exekveringen.

Nuvarande status

För detta projekt finns en relativt utförlig beskrivning i DLU 74, kap 6a. SIMLISP består av ett antal moduler enl fig



Interpretatorn skrevs under 1973 och är redovisad i DLU 74.

Under 1974 har förprocessorn skrivits och uttestats. Förprocessorn får som input ett Simula-program representerat i S-notation på enklaste sett. Förprocessorns uppgift är sedan att i princip utföra allt som går att göra under en motsvarande kompilering av Simulaprogrammet. Bland dess uppgifter kan nämnas,

- identifierar-substitution. (Anteckna till vilket block varje identifierare tillhör.)
- läges-sökning. (Associera varje läges identifierare med en "väg-beskrivning" som anger hur man från blocket kan till själva läget.)
- syntaxcheck
- typcheck (Spec ref variabler)
- block-konkateneringar (Inklusive inner och virtual)

Det visade sig bl a, att förprossorn var betydligt jobbigare att skriva än Interpretatorn. Främsta anledningen till detta var Simula:s "dubbla" textomgivning dels via block-nivåer, dels via konkatererade klasser.

Förprocesson är skriven enligt principen varje Simula-ord (begin real inspect etc) har ett antal procedurer associerat till sig.

En allmän funktion execute (form, type) exekverar en form, och om form är ett Simula-ord uttryck

ex: (BEGIN decl satser), (REAL X Y) etc.

används den procedur som type anger, och denna appliceras på argumenten.

Den vanligaste typen är SUBSTITUE, som anger till vad ett Simula-uttryck skall substitueras (alla Simula-ord har SUBSTITUTE-procedurer). Dessutom finns sådana som DECL (sköter om deklarerande egenskaper), CLASSATTRIBUTE (~ som DECL), REF (ger ref-värdet av ett uttryck. Används vid typ-kontrollen).

För att illustrera tekniken visas hur inspect-satsen översätts.
SIMULA:

a) inspect oe when classid$_1$ do s_1
when classid$_2$ do s_2
:
otherwise s_n;

b) inspect oe do s_1 otherwise s_2;

Satserna s_i skall utföras, som om de tillhörde en viss klass. I a-fallet ges klassen av $classid_i$, i b-fallet ges klassen av hur oe är kvalificerad.

ex: om oe är X, och X deklarerad ref(A)X;

är X kvalificerad till A oberoende av vad X pekar på . Vid variabelsubstitutionen som skall utföras, måste denna textomgivning alltså simuleras.

In-översättaren har översatt till

a) (INSPECT oe (WHEN $classid_1$ S_1)
(WHEN $classid_2$ S_2)
⋮
(OTHERWISE S))

b) (INSPECT oe (DO S) (OTHERWISE S))

Förprocessons kod för inspect-satsen:

```
SUBSTITUTE inspect (oe . x);
            substinspect (execute (oe, 'SUBSTITUTE),x)|
```

Definition av SUBSTITUTE-proceduren för inspect. (oe . x) betyder att
oe skall bindas till 1:a argumentet
x skall bindas till resten av argumenten.
| anger slut på procedur-definition.
Definitionen säger oss bl a att oe-uttrycket först skall substitueras (execute-anropet).

```
Substinspect(newoe, l); prog(quaoe);
    if newoe then quaoe := quaof(newoe);
                           mkinspect(newoe,
                               executelist(l, 'INSPECT)|
```

Denna procedur anropas direkt från den förra. newoe är det substituerade "object expression", l är lista av WHEN, DO eller OTHERWISE-satser. Om newoe ≠ NIL (=NIL om error inträffat under substitueringen) beräknas först dess kvalificerade värde (REF-värdet) av proceduren quaof (ej med-

tagen här). quaoe används som fri variabel i INSPECT -do nedan. Därefter exekveras INSPECT-procedurerna i listar l, och en färdig inspect-sats görs av mkinspect, som är def som.

```
mkinspect (newoe, newl); ('INSPECT, newoe, newl)|
```

Återstår att se vilka INSPECT-procedurer som DO, WHEN och OTHERWISE har.

```
INSPECT when (classid, s); prog (newclassid)
   newclassid := getlexbind(CLASSID);
   newblock newclassid do
            mkwhen(newclassid,        )
                   execute(s, 'SUBSTITUTE)|

mkwhen(newclassid,news);
       list('WHEN, list('CHECKWHEN, newclassid), news)|
```

getlexbind är en rutin som hämtar upp nya identifierare värdet i en viss angivning.

Konstruktionen newblock id do s är ett anrop till en procedur som uppdaterar den lexikaliska omgivningen genom att push-a id. id förutsätts vara namn på ett block. Proceduren newblock är så pass viktig, så anrop till denna har ersatts med ovanstående konstruktion för att anropen skall synas bättre i koden. I den nya omgivningen undersöks sedan satsen s cch mkwhen anropas. Lägg märke till att den lista mkwhen konstruerar, innehåller en konstruktion som under run-time blir ett anrop till test funktionen checkwhen.

```
INSPECT do(s);
   newblock quaoe do list ('DO, execute (S, 'SUBSTITUTE)|
```

Även do-klausulen uppdaterar den lexikaliska angivningen innan satsen s undersöks. Här bestämms nya omgivningen av quaoe, som är en fri variabel från proceduren substinspect. quaoe's värde är det klassnamn 'objekt expression' i inspect-satsen är kvalificerad till.

```
INSPECT otherwise(s);
    list('OTHERWISE, execute(S,'SUBSTITUTE))|
```

Otherwise-klausulen behöver bara returnera ett otherwise-uttryck, där satsen s är genomgången.

7. MANIPULATION AV FORTRAN-PROGRAM

7a. SUGFOP - syntaktisk socker för Fortran

SUGFOR är en språkutvidgning och en preprocessor för utvidgade kontrollstrukturer och annat syntaktiskt socker i Fortran. SUGFOR utvecklades under första kvartalet 1974 vid DLU i samarbete med Tom Smedsaas från avdelningen för numerisk analys. Utvecklingen gjordes av två skäl. Dels planerades ett antal större projekt som skulle innebära programmering av stora Fortranprogram för icke-numeriska problem, vilket alltså avsevärt skulle underlättas om utvidgade kontrollstrukturer fanns att tillgå, och dels gjordes utvecklingen av språket och preprocessorn för att utröna hur mycket arbete en sådan språkutvidgning innebär och om syntaktiskt socker lagt ovanpå ett existerande programmeringssystem kan förbättra tillförlitligheten i program skrivna med hjälp av språkutvidgningen. Arbetet med språkutvidgningen och preprocessorn krävde sammanlagt två manmånaders arbete.

SUGFOR finns utförligt beskrivet i DLU 74/31 (SUGFOR: Syntactical Sugar for Fortran) men en kort genomgång av språkutvidgningen görs nedan. Därefter beskrivs några av de viktigare erfarenheterna från arbetet med SUGFOR.

Principer för språkutvidgningen

Det stod redan från början klart att preprocessorn för SUGFOR inte skulle behöva göra en fullständig analys av Fortran-delarna av den källkod den skulle behandla. Skälet till detta är att en fullständig analys av Fortran är ett "grötigt" arbete som kräver mycket kod för att ta hand om alla specialfall och konstiga konstruktioner som förekommer i Fortran. Speciellt besvärligt är att ta hand om de kontextberoende konstruktionerna. Dessutom ställer Hollerithkonstanterna till besvär. (För den som inte tidigare försökt analysera Fortran ges här några exempe på vackra konstruktioner att analysera:

```
DO10I=1,3          do-loop
DO10I=1.3          tilldelningssats
DO10I=2H,3         tilldelningssats (i Fortran för CD3600)
REAL*8HOLLER       ej Hollerithkonstant (IBM Fortran IV)
DATA A/8HOLLER  / Hollerithkonstant )
```

Om preprocessorn inte skall göra en fullständig analys av kälkoden måste den kunna känna igen nyckelord och operatorer direkt i input-strängen. Detta medför t ex att blanka måste införas kring nyckelord som inte omges av operatorer. Det har också varit nödvändigt att ta bort Hollerithkonstanten och endast tillåta strängar omgivna av "blippar" ('). Dessutom har i språkutvidgningen använts operatorer som inte ingår i Fortran, t ex semikolan.

Det utvidgade språket består dels av särskilda SUGFOR-satser och dels av vanliga Fortransatser. SUGFOR-satserna innehåller också delar av vanlig Fortran-kod, t ex villkorsuttryck, som kopieras in på rätt ställe i den genererade koden. Fortransatserna kopieras direkt till denna utan ändringar.

Kontrollstrukturer

Alla de vanliga kontrollstrukturerna i Fortran har tagits bort och därmed också satsnumret. Detta har ersatts med en alfamerisk label som får stå före vilken sats som helst i programmen. GOTO-satsen har alltså behållits men har gjorts till en SUGFOR-sats med alfamerisk hoppadress. De kontrollstrukturer som införts i stället för de borttagna Fortransatserna är starkt inspirerade av programmeringsspråket MARY, som i sin tur är inspirerat av Algol-68. Största skillnaden gentemot motsvarande i t ex Algol är att varje sats även har ett avslutande nyckelord (t ex FI) som gör det möjligt att ha följder av satser inuti kontrollstrukturerna utan användning av särskilda satsparenteser.

De kontrollstrukturer som införts är en valsats (IF-sats), en villkors-loop (DO-loop) och en stegloop (FOR-loop). IF-satsen har THEN-gren och en ELSE-gren som kan utelämnas. Satsen avslutas alltid med FI. Villkors-loopen kan ha villkoret placerat antingen före eller efter loopen och rakt villkor (WHILE) eller negerat (UNTIL) är tillåtna på båda ställena. Loopen är alltid omgiven av DO....OD. FOR-loopen tillåter stegning antingen uppåt eller nedåt och har ett implicit steg på +1 eller -1 om inget annat anges. Loopen omges även här av DO....OD.

Fortrans subrutin- och funtionsstruktur har bibehållits oförändrad.

Andra utvidgningar

Det utvidgade språket skrivs i fritt format med satserna åtskilda av semikolon. Särskilda satsparenteser finns men är ej nödvändiga i någon konstruktion. En enkel makro-möjlighet har införts och dessutom en "automatmakro" som expanderar namn som börjar med tecknet et (&) till ett unikt namn på sex tecken. Automatmakron kan användas för att tillåta namn på variabler.

Förprocessorn

Förprocessorn är skriven helt i Fortran. Den består av en scanner med makroprocessor, en syntaxanalysator med semantiska rutiner och ett antal symboltabeller. Syntaxanalysen görs med "recursive descent"-metoden i en internt rekursiv Fortranrutin. Konstruktionen av utdatakoden görs av de semantiska rutinerna och koden skrivs ut av ett särskilt rutinpaket. Hela förprocessorn är starkt modulariserad och uppdelad i funktionella enheter.

Erfarenheter

SUGFOR har ännu inte använts i något större programmeringsprojekt. Detta medför att någon objektiv bedömning av hur mycket bättre eller sämre det är att koda i SUGFOR än i Fortran inte kan göras. Trots detta kan flera påpekanden göras.

För det första måste man inse att en språkutvidgning som SUGFOR inte på något sätt ökar den interna säkerheten i ett programmeringssystem, och att de flesta farligheterna i Fortran kvarstår i SUGFOR. Det bör emellertid vara enklare att överföra algoritmer till SUGFOR än till Fortran tack vara de utvidgade och mera välstrukturerade kontrollstrukturerna. SUGFOR-progran bör också bli något lättare att läsa.

Man måste också inse att det alltid uppkommer svårigheter när ytterligare en nivå läggs mellan programmeraren och exekveringen av hans program. Felmeddelanden och annan information blir ännu svårare att härleda till rätt ställe i källkoden och felsökningen försvåras. Det tillkommer också ytterligare en felkälla i och med att programmet behandlas av förprocessorn som i speciella fall kan generera felaktig kod hur väl den än är testad.

Slutsatsen måste bli att SUGFOR var ett nyttigt experiment men att system av denna typ visserligen kan resultera i förbättringar av programmeringssystem men att de inte kan ge så stora totala förbättringar av systemen att det lönar sig. Däremot kan naturligtvis förprocessorbaserade språkutvidgningar vara lämpliga för speciella tillämpningar, t ex understöd för hantering av särskilda datastrukturer.

7b COMLIST

Som trebetygsarbete har framtagits ett program för att generera globala korsreferenslistor för variabler i COMMON-block i Fortranprogram. Programmet kan för ett visst COMMON-block visa hur detta är definierat i olika rutiner. Det kan också visa vilka COMMON-block som är definierade i olika rutiner. Den mest avancerade funktionen är att för en given plats (ord, byte etc) i ett COMMON-block visa hur just denna är använd i olika rutiner.

Användningen av COMMON-block i Fortran kan ställa till problem eftersom samma block kan definieras olika i skilda rutiner. Detta kan medföra att samma fysiska minnesplats kan vara använd som heltal i en rutin, som flyttal i en annan etc. Möjligheten till korta and långa heltal och flyttal ökar riskerna för fel ytterligare. Dessutom finns i Fortran möjligheten att med EQUIVALENCE-satsen utnyttja samma minnesplats för olika variabler i samma rutin. Detta medför att det kan vara mycket svårt att i stora program veta till vad och var en viss variabel egentligen används, något som är ett särskilt stort problem vid modifiering av ofullständigt dokumenterade program. COMLIST-programmet är avsett som ett hjälpmedel särskilt för sådant arbete.

P R O G R A M R E D S K A P O C H M E T O D E R

8: 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 som denna gemensamma bas.

Under 1974 har situationen varit sådan att vi dels haft tillgång till Uppsala Datacentrals maskin, där det nyutvecklade INTERLISP-systemet använts, 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, m m) för många projekt varit mer attraktivt att köra på än UDAC:s 370/155.

Vi har därför under året fortlöpande haft kompatibilitetsproblem mellan 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 finna vissa program på Simon eller LISP F1 men ej under INTERLISP.

Datalogilaboratoriets programbibliotek beskrevs närmare i förra årsrapporten (DLU -74).

9. MANIPULATION AV LISP-PROGRAM

9a REDFUN

REDFUN är ett program, som utför viss optimering på LISP-kod:

a) Direktevaluering. T ex 2 + 7 blir 9.

b) Partialevaluering.
T ex "if true then A else B", där A och B är godtyckliga uttryck, blir "A".

c) Förenkling.
T ex car[list[a,b,c]] blir a.

d) Lambda-expansion, dvs insättning av procedurdefinitioner med formella parametrar substituerade mot aktuella på anropets plats.

Ursprungligen skrevs REDFUN för att användas i samband med PCDB och har beskrivits utförligt i tidigare årsrapporter. Programmet verkade dock även kunna användas som "generell kompilator", dvs för att specialisera parameterstyrda program där någon eller några parametrar är givna på förhand. Under 1974 gjordes en del inledande försök med att applicera REDFUN på GUP-programmet. GUP är ett program för utskrift av LISP-databaser och det har ett mycket stort antal parametrar. De flesta parametrarna får i allmänhet defaultvärden.

Eftersom REDFUN ursprungligen var avsett att operera på det subset av LISP, som genereras av PCDB och GUP är programmerat "fritt ur hjärtat", visade det sig att REDFUN måste modifieras i vissa avseenden:

En ny huvudoperation tillfördes till de tidigare nämnda. Vissa procedurer specialiseras nu av REDFUN och ges nya namn, så att i princip genererar REDFUN flera specialiserade procedurer av en generell.

Administrationen av tilldelning av värden till variabler visade sig vara relativt komplicerad. Om en tilldelningssats förekommer inuti en villkorssats där REDFUN inte kan veta vilken gren som ska evalueras, kan REDFUN i allmänhet inte veta något om variabelns värde efter villkorssatsen.

Å andra sidan skulle REDFUN inuti villkorssatsen kunna utnyttja viss kunskap om variabeltilldelningar. Detta problem löstes genom att REDFUN får hålla reda på variabeltilldelningar och inträden i och utgångar från villkorssatser.

Sedan dessa och andra problem lösts, testades REDFUN på utskrift av en del av SNV-databasen (13a) med hjälp av GUP. Effekten av specialiseringen illustreras av nedanstående tabell:

	Ursprunglig GUP	Specialiserad GUP
Storlek	2150	1599
Antal procedurer	29	22
Tid för utskriften (Interpreterad kod)	68823 ms	7668 ms
Tid för utskriften (Kompilerad kod)	5161 ms	2220 ms

En sammanfattning av de resultat och erfaranheter vi fått av detta arbete med REDFUN har dokumenterats i en rapport:

Beckman, Haraldson, Oskarsson och Sandewall. A partial evaluator and its use as a programming tool (DLU 74/34).

9b REDCOMPILE

Programmet REDFUN, beskriven i föregående avsnitt, används för att optimera LISP och används då vi vill specialisera (skräddarsy programmet för en applikation) ett generellt program. Under 1974 har vi arbetat på en "kompilator" för REDFUN, som vi alltså kallar REDCOMPILE. Idée är att köra det generella programmet genom REDCOMPILE och samtidigt ange vilka parametrar, som blir konstanta då det generella programmet används för en speciell tillämpning. REDCOMPILE skall nu generera ett nytt program av det generella programmet, som är så utformat att det nya programmet direkt genererar det specialiserade programmet. I förra årsrapporten (DLU -74) beskrevs detta resonmang på ett mer formellt sätt.

Låt oss med ett enkelt exempel se vad som händer. Antag vi har det generella programmet

```
(LAMBDA (A B P Q)
(SELECTQ P
   (1 (FOO (CONS (CAR Q) A) B))
   (2 (FIE (CONS B (CDR Q)) A))
   (FUM A B)))
```

Vi antar vidare att P och Q är parametrar, som kommer att vara konstanta och ha ett känt värde under den specialiserade tillämpningen.

Om nu P har värdet 2 och Q värdet (K L M N) kommer REDFUN att kunna optimera programmet till

```
(LAMBDA (A B)          (*)
(FIE (CONS B '(L M N)) A))
```

vilket alltså är det önskade specialiserade programmet.

Andra metoden att genera samma program är nu att först med REDCOMPILE generera en programgenerator. REDCOMPILE får som argument det generella programmet samt en lista på parametrarna P och Q. Resultatet från REDCOMPILE blir

```
(LAMBDA (P Q)
(SELECTQ P
   (1 (LIST 'FOO (LIST 'CONS (KWOTE (CAR Q)) 'B)))
   (2 (LIST 'FIE (LIST 'CONS 'B (KWOTE (CDR Q)) A))
   (LIST 'FUM 'A 'B)))
```

Med P = 2 och Q = (K L M N) kommer detta program att generera det specialiserade programmet (*).

Den senare metoden blir ju bättre om vi vill generera flera olika specialiserade program för samma uppsättning parametrar.

REDCOMPILE har under året utvecklats så att det i stort klarar samma uppgifter som REDFUN. Detta innebär att funktioner som är PURE (ex car och cons) kommer att evalueras om argumenten är kända. För funktioner, som är OPEN, kommer funktionskroppen att sättas in i koden. Funktioner av typ SPECIAL (ex cond och selectq) kan också bearbetas. Däremot har programmet besvär med funktioner med sidoeffekter (ex setq) och prog-uttryck.

Användningen av REDCOMPILE har under året främst varit inriktad på PCDB. Smärre ändringar har utförts i PCDB för att REDCOMPILE skulle klara av dess kod. Vissa specialfunktioner för att underlätta användningen av REDCOMPILE för PCDB har skrivits, exempel en funktion compiledefiner, som används för att redcompilera en PCDB definer. Programmet finns upplagt i DLU's programbibliotek för INTERLISP.

9c INTERLISP simulator

Inledning. Under sommarskolan i Datalogilaboratoriets regi i juni 74 skulle bl a ges undervisning i och tillfälle att köra LISP. Det var önskvärt att endast behöva lära ut en LISP-dialekt nämligen INTERLISP. Vi hade tillgång till två LISP-system:

1. INTERLISP på Uppsala Datacentrals IBM-370, detta system kunde vi inte utnyttja interaktivt i någon större utsträckning (endast ibland på kvällstid).
2. LISP 1.6 på Stockholms datacentrals DEC-10, detta system var tillgängligt interaktivt dygnet runt.

För att ge kursdeltagarna möjlighet att köra interaktivt, vilket ger större förståelse för LISP:s fördelar, och snabbare inlärning, behövdes en modifikation av LISP 1.6-systemet. För att kunna utnyttja de program som redan fanns skrivna i LISP 1.6-dialekten, skulle systemet acceptera både INTERLISP och LISP 1.6. Denna modifikation (Interlisp-simulatorn) gjordes under maj månad med stöd av Sekretariatet för nordiskt kulturellt samarbete. INTERLISP-simulatorn består av ett antal filer som laddas in till LISP 1.6, det resulterande systemet kallar jag i fortsättningen Interlisp-systemet.

Några olikheter mellan INTERLISP och LISP 1.6

- funktioner måste anropas med korrekt antal argument.
- annorlunda "värdeuppslagning" (shallow-binding) gör att globala värden ej är åtkomliga.
- filhantering är mycket olika.
- trace är ful och oöverskådlig (ingen indentering).
- editor saknas.

- i felhanteringen saknas break och backtrace.

- annorlunda funktionsuppsättning.

Beskrivning av implementering

Alla funktioner blir när de definieras översatta så att funktionen klarar anrop med för få eller för många argument. Funktionskroppen(arna) översätts inte. Översättningen åstadkommes genom att definiera om DE och DF, som lägger översättningen under indikatorn EXPR eller FEXPR. Den ursprungliga definitionen sparas under indikatorn PP.

För att användaren inte ska förvillas av att funktionen har ändrat utseende verkar "titt-funktionerna" (PP och EDITF) på PP-egenskapen. När man editerar ser man alltså det man själv skrivit, och när man går ur editorn görs en ny översättning av den ändrade funktionen.

De funktioner med samma namn, som fungerar olika i INTERLISP och LISP 1.6 är omdefinierade till INTERLISP-varianten, när denna är en generalisering. Det fordras t ex att man inte utnyttjar att AND returnerar T, utan att det fungerar med vad som helst ≠ NIL, för att LISP 1.6-program ska gå att köra i INTERLISP-systemet.

För den som vill slippa funktionsöversättningen finns funktionerna __DE och __DF, som fungerar som DE och DF i LISP 1.6.

Begränsningar hos Interlisp-systemet

1. De funktioner som ingår i Interlisp-systemet måste anropas med korrekt antal argument. (De funktioner som man definierar själv har INTERLISP:s variabelbindning). Anrop med för många eller för få argument ger felutskrift om funktionen ej är kompilerad. Om funktionen är kompilerad fås ingen felutskrift, men "överblivna" lambda-variabler binds ej till NIL.

2. I CAR av en atom ligger den senaste bindningen, ej det globala värdet. RPLACA på en atom fungerar som SET. Detta kan orsaka fel om man vill ha tag på eller ändra det globala värdet på en atom, som används som lambda- eller prog-variabel.

3. Felhanteringen är mycket olik INTERLISP:s. Break och backtrace saknas.

4. Funktioner för hantering av strängar och arrayer saknas, eller fungerar annorlunda.

5. GET, PRINT och SUBST fungerar som i LISP 1.6 dvs GET fungerar som INTERLISP:s GETP. PRINT gör radframmatning före utskrift. SUBST har omkastade argument (man kan använda DSUBST).

6. Tecken:

i st för	använd	
'	Ö eller	(quote)
<	[	(superparenteser)
>	]	
%	/	(escape karaktär)

7. Använd FUNCTION (ej QUOTE) på "funktionsargument" för att få INTERLISP:s variabelbindning, och möjlighet att ha flera funktionskroppar.

Editor

Toppfunktioner:

EDIT, EDITF, EDIFP, EDITV.
Editorn promptar med E:
Kommandon som fungerar som i INTERLISP:
(n och m betecknar heltal)
?
P
PP
n
(n S1 S2 .. Sm)
(n)
(-n S1 S2 .. Sm)
(R S1 S2)
(LO n)
(LI n)
(RO n)
(RI n m)
(BO n m)
(BI n m)
(BI n)
(N S1 S2 .. Sm)

Kommandon som ej fungerar som i INTERLISP:
O och UP fungerar på samma sätt, men går inte upp i strukturen på samma sätt som INTERLISP-editorn, (det kan gå hackigt efter F-kommandon.)

UNDO saknas, men man har chans att ändra sig efter det att man gjort OK (exit). Då frågar nämligen editorn: CHANGE DEF? om man då svarar NO återställs den gamla strukturen.

Man har ännu en chans att ändra sig om man har editerat en funktion: gör (UNSAVEDEF fn) när som helst efter editeringen. F går aldrig upp på högre nivåer efter det att något har hittats. "E uttryck", evaluerar uttryck. Om man utesluter "E", så uppfattas troligen "uttryck" som ett felaktigt kommando.

Dessutom finns "PR n" som printar till nivå n.

Filhantering

(MAKEFILE ÖFOO) skapar filen FOO.L0 i ditt bibliotek. "0" i "L0" är ett generationsnummer. Nästa (MAKEFILE ÖFOO) skapar filen FOO.L1 osv upp till värdet av atomen MAKEFLIMIT (initierad till 10). Om du försöker göra ytterligare (MAKEFILE ÖFOO) får du felutskrift. Då kan du antingen ändra värdet på MAKEFLIMIT till ett tal större än 10, eller spara körningen och scratcha eller ändra namn på filer och sedan göra om makefilen.

(LOAD ÖFOO) laddar den fil som har högst generationsnummer av de som har generationsnummer≤MAKEFLIMIT. Det kan alltså vara farligt att skapa filer med högre generationsnummer än 10.

(LOAD Ö(FOO.L4)) laddar FOO med generationsnummer 4.
(MAKEFFILE ÖFOO ÖFAST) gör en icke-prettyprintad fil.
(MAKEFILE ÖFOO) gör en prettyprintad fil.

Den pretty-printade filen är endast något större, men tar flera ggr mer tid att skriva ut.

LOAD (eg DSKIN) skriver ut värdet på varje evaluerat uttryck. För att undvika de ofta långa listningarna gör MAKEFILE en PROG omkring hela filen.

Denna mekanism kan stängas av (t ex kompilator fordrar detta), genom att sätta variabeln PROGFLAG till NIL (den är initierad till T).

Program blir omkring 2 ggr långsammare i Interlisp-systemet än motsvarande LISP 1.6-program. Detta beror till stor del på att de frekvent använda funktionerna CAR och CDR är omdefinierade och ej kompilerade. En annan sak som slöar till program är den extra kod i varje funktion (läggs till vid funktionsöversättningen) som ska ta hand om variabelbindningen. Interlisp-simulatorn har använts intensivt under de två veckor sommarskolan varade, och har visat sig vara pålitlig om man inte försöker sig på något "extra".

9d TRANS, ett paket för översättning från INTERLISP till LISP 1.5

På Datalogilaboratoriet sker nu den mesta programmeringen i INTERLISP. TRANS skrevs för att kunna lämna över program, som skrivits i INTERLISP, till personer som bara har tillgång till ett LISP 1.5-system. Samtidigt var det en test på PMG (program-manipulator-generator se DLU 74/1). TRANS består av:

1. topfunktion som har som argument en lista på de funktioner, som ska översättas.
2. 3 centrala genomsökningsfunktioner, som är genererade av PMG.
3. ett antal MAKROn och EMAKROn, som beskriver de ändringar som ska göras i koden vid översättningen.

TRANS ger under översättningen följande utskrifter:

för varje funktion

1. en lista på de funktionsanrop, som har ändrats (ej endast namnet på funktionen utan även argumentordningen kan ändras).
2. en lista på de funktioner som var odefinierade (behandlas av TRANS som eval-funktioner).
3. en lista på de INTERLISP-funktioner som måste definieras för att översättningen ska gå att köra på LISP 1.5-systemet.

efter hela körningen

en lista på de av användarens noeval-funktioner som saknar MAKRO-definition (det finns risk för felöversättning av dessa anrop).

TRANS konstruerades ursprungligen för översättning till LISP F1. En stor översättning har gjorts, nämligen av PCDB, (ett stort paket ca 375 funktioner). Den översättningen var till ett annat slags LISP 1.5-system (med stack). Vi fick veta vilka funktioner systemet innehöll, och skrev utgående från detta några nya MAKROn. Det var den enda ändring som behövdes. Det är alltså lätt att modifiera TRANS för översättning till andra LISP-system.

10. Programredskap för informationsstrukturer på lägre nivå

10a. PCDB - Predicate Calculus Data Base

PCDB-paketet förutsätter att användaren beskriver problemomgivningen i predikatkalkyl. Han skall deklerera 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 -74) och inriktningen på arbetet har sedan dess inte ändrats. Den intresserade läsaren hänvisas därför till dessa årsrapporter.

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 René Reboh. Under hösten 1972 påbörjades arbetet med en ny reviderad version (C-versionen) och detta arbete har fortgått sedan dess och har utförts av Anders Haraldson.

Arbetet under 1974 har främst varit inriktat på att testa systemet och kanske främst att testa REDFUN och REDCOMPILE på PCDB. Vissa problem har uppstått och ändringar har fått göras i PCDB koden. Vissa av problemen beskrivs i den nya rapporten (DLU 74/34). Visst underhåll, ändringar och effektivisieringar har utförts under året. Bland annat kan nämnas att det har införts "handtag" i vissa av de övervakande funktionerna för slutsatsdragningen, så att användaren själv kan komplettera med egna funktioner. Detta ger bl a möjligheten att lägga in en meritfunktion för val av subfråga.

10b DABA - Ideas about management of LISP data bases*

The trend toward larger data bases in A.I. programs makes it desirable to provide program support for the activity of building and maintaining LISP data bases. Many techniques can be drawn from present and proposed systems for supporting program maintenance, but there are also a variety of additional problems and possibilities. Most importantly, a system for supporting data base development needs a formal description of the user's data base. The description must at least partly be contributed by the user. The paper discusses the operation of such a support system, and describes some ideas that have been useful in a prototype system.

LISP is really a programming language for a certain type of data bases, and it is less interesting as a 'list processing' or 'list structure' language. The item in LISP that makes it different from the other major programming languages is not the cons cell, but the atom. Atoms are the carriers of property-lists, which is the primary representation in LISP's data base. Atoms are also essential for the facility to read and write data structures, which is what makes LISP a good interactive language. Similarly, the most significant built-in functions in LISP are not car, cdr, and cons, but get, put, and other functions for accessing property-lists in the LISP data base. The language also contains some functions of secondary significance which can be used to construct and decompose properties, for example cons, maknam, cdr, and explode.

Such an alternative view of our favourite programming language is becoming increasingly useful with the present trend toward larger and more complex data bases in Artificial Intelligence systems. It is also supported by the fact that, with the on-going blurring of the program/data distinction, programs become integrated parts of the data base in a less trivial sense than used to be the case. A number of new aspects of the programming language and its use arise when the focus of interest is changed to the data base, for example:

- Current programming practice for using LISP's representational primitives. The structure of atoms and lists is quite different from the record structure of other languages "with data structures",

---

* This work has been done at MIT and its LISP version, MACLISP, is used.

and encourages a different methodology. This methodology already exists, but it should be talked and written about, both for tutorial purposes, and as a basis for developing it further.

- Block structure in the data base. It is common practice to organize LISP programs into "blocks", or groups of closely related functions. Such block structure encourages modularity and facilitates maintenance. Both purposes would be worthwhile for the data base as well. However, a number of new and interesting problems arise when one attempts to organize the data base into blocks, for example because data items can be related in multiple ways, so that clustering criteria are not trivial to decide.

- Self-description in the data base, whereby the data base contains a description of itself, and (in more developed systems) of its relation to the intended application. Such self-description could be made useful both for the user (as a documentation aid), and for programs which use it as parameters, to determine how operations on the data base are to be performed.

- Utility programs for LISP data bases. By utility programs, I mean general programs which are primarily intended to be called directly by the user, rather than as subroutines from another program, and which do some service operation on a data base. Utility programs for programs are in common use, for example compilers, editors, file grinders, and (to some extent) indexers. Many of these operations generalize to data bases as well. Others can be added, for example consistency checks.

Since programs is a special case of data in LISP, one can often extend the use of programs that were written for operating on programs, to be used for other types of data as well. But it is much better to design such programs right from the start for data bases in general.

With data base techniques in LISP as my present major interest, I have been toying for a while with the idea of a <u>support system</u> for data base management in LISP. Such a system would be a collection of programs (and associated data) which support the user in his work with the data base. The term "data base" is here taken to mean collections of knowledge that are used by one or more programs. It does not refer to temporary data such as hypotheses or sets of subgoals, which are created

during a computation and later garbage collected, or discarded at the end of the computer run. Correspondingly, the "user" whom the system supports, develops not only programs, but also data bases in the given sense. The support system could develop into a "data base hackers assistant".

An experimental system, called DABA, has been instrumental in developing and testing some of the ideas, and hopefully will also serve as an illustration of them. DABA is a MACLISP program. This working paper is an attempt to summarize my ideas at present. For concreteness, it uses some of the notation of the DABA system, but it is not a systematic description of DABA.

The major service that a support system can offer its user is utility operations. Sometimes the user will be writing down parts of a data base directly, much like he writes down a program. He then needs the same kinds of utilities as for program development, which enable him to administrate and update easily. At other times, the user will obtain parts of his data base from computation. They may be the accumulated experience of a performing program, or the result of running a utility program on previously existing data (for example a program for shift of representation). In either case, the user needs a program which can administrate the new data, shovel them around, and integrate them properly in his data base.

Unfortunately, the user can not count on obtaining such a service without effort from his part: he has to specify his representation and data-base structure to the utility program. (Even if the data base contains information about itself, he at least has to specify what conventions he used for the self-description). Therefore, a support system for utilities must necessarily contain a system for description of data bases. Ideally, one would like to have general-purpose descriptions, which can be used by all utilities, and one would also like to store the description in the data base (which was called self-description above), so that the utilities can be applied recursively to the descriptions.

A user supporting system should of course allow for the variety of different representations that are found in current LISP programming. Some LISP users work directly with the data base primitives provided by the language, but many develop their own higher-level representations, or

or use available systems such as PCDB, or the data base handlers in Conniver or QA4. A support system should therefore enable one to make a definition of for example Conniver's data base primitives (such as contexts) in terms of the underlying LISP primitives, so that the user then can talk in Conniver terms when he describes his Conniver data base, and when he calls for a utility operation on it.

A system for describing the representation in a part of a data base, could also be used to describe a program with respect to what representation it assumes in its data. A program which adjusts data to fit given programs, or vice versa, is then a desirable utility.

The requirements of utility programs actually call for two different kinds of self-description. There is self-description of representation, where the syntax and perhaps also semantics of the chosen representation are specified. But many utilities can be characterized as scanners, in the sense that they scan over a specified segment of a data base, and perform some operation throughout it. Examples of scanner utilities are for saving data bases on files, for presentation (as a generalization of prettyprinting), for checking, and for shifts of representation. Scanner utilities then need a description of extent of parts of the data base. It is not yet clear to me how one should properly handle the relationship between description of representation and description of extent, but a tentative model is described in the present paper.

One part of the description of a representation for a data base is the set of procedures which access (in the sense of both 'get', 'put', 'modify', and 'delete') data bases that use the representation. (Too often the access procedures are the only description). The DABA system assumes that access procedures are part of the self-description. This means that an application program can access the data base through DABA, which knows where in the self-description the access function is located. For efficiency of computation, one will often want to open-compile such calls and eliminate the detour through DABA, but the idea of first storing the access function in the description has advantages, for example that it becomes simpler to generate and retain access procedures from other parts of the description.

Generation and default definition of access functions are achieved in the prototype DABA system by a recursive access mechanism: in order to use the data base, an access function is retrieved and used, and at least in principle the access function is retrieved in the same way, using its access function, and so on. Such recursive access also provides a simple and elegant basis for handling other features in the system, for instance description of data blocks. It has some obvious efficiency problems, which I think however can be resolved. The next section describes the access mechanism in more detail.

It is attractive to let the descriptions of the data base be in the data base itself, so that the support system can be used recursively. This necessitates a choice of data representation for phrasing the description in, but should not and need not imply a choice or a restriction of what data representations can be described. I have preferred to use an object/property-type structure for the descriptions, rather than for example a relational structure, since the former is the closest to the property-list data base primitives in LISP system, and since the auxiliary systems for alternative and higher-order representations are ultimately defined from such primitives. The object/property representation is augmented with the well-known method of nested property-lists. With this representation, for each pair of carrier and indicator, the data base may contain a property, which may be an arbitrary entity, but in particular may be a set of assignments of sub-properties to sub-indicators.

A blocking concept as discussed above has also been useful for structuring the descriptions. The data base is viewed as a collection of "items" (which may be property assignments, relations, some variety of frames, or something else that is an entity), and such items are grouped into blocks. Some of the possible connotations of that term are however not intended: no parenthesis-like nesting of blocks and no scope concept for identifiers are being used. The term "data set" would have been more appropriate if it had not already been taken for another purpose. The primary intended analogy is with the practice to group sets of functions in large LISP programs into "files".

Blocks have some resemblance to contexts in Conniver and QA4, except that contexts are mostly intended for "scratchpad" data. Like for contexts, the same carrier/indicator pair may have different properties in different blocks.

Blocks proved useful for structuring the data base descriptions, both the descriptions that the DABA system expects the user to provide, and the higher-level descriptions in the system itself. I believe that data base block structure can also be very useful in many applications. One particular usage is for conserving storage in LISP systems which are plagued by space problems in the heap: blocking enables one to keep little-used parts of the data base as text files, and only bring them in when needed.

10c TOP

Vid all kommunikation mellan människan och datorn behövs en översättare, en parser. TOP är en sådan parser. Så snart människan ska mata in något i datorn som har mera struktur i sej än bara enstaka symboler, så är TOP mycket bra att ha. Speciellt lämpat är TOP om man vill mata in formeluttryck eller programstrukturer med en Algolliknande form.

TOP är ett mycket litet programpaket. Därför är det mycket lätt att utvidga och modifiera. TOP arbetar i huvudsak så att varje objekt i inputströmmen läser in sin del av inputen. TOP är en ren vänster till höger-parser, alltså ingen backup sker under parsningen. Detta gör TOP även snabb och effektiv.

TOP kan skrivas i varje programmeringsspråk, som tillåter rekursiva funktionsanrop. Sedan är det också bra om programmeringsspråket innehåller en eval-facilitet, så att man kan utvidga syntaxen medan man använder TOP. Språken LISP och REC uppfyller båda dessa krav, och nedan anges hur TOP ser ut i dessa båda språk.

TOP är även ett sätt att beskriva en grammatik för ett språk. Men till skillnad från en BNF-beskrivning, så anger TOP villka uttryck som är OTILLÅTNA, medan en BNF-beskrivning i stället anger vilka uttryck som är TILLÅTNA.

TOP:s arbetssätt

För att kunna använda TOP behöver man en SCANNER som delar upp inputströmmen på önskat sätt. Denna funktion heter RDOBJ, och den sparar senast inlästa objekt i den globala variabeln OBJ.

TOP läser in en struktur. Vad som är en struktur beror på den grammatik man har. Har man en grammatik för formeluttryck, så är en struktur det, som anges av följande exempel på skillnaden mellan ett objekt och en struktur:

"a"	är ett objekt och en enkel struktur.
"abc"	är likaså ett objekt och en enkel struktur.
"("	är ett objekt men ej en struktur.
"a+"	är varken ett objekt eller en struktur.
"a+b"	är en sammansatt struktur bestående av tre objekt.
"(a+b)"	är en enkel struktur bestående av fem objekt.

Varje objekt har i varje sammanhang en konstruktionsfunktion. Ett objekt kan ha olika konstruktionsfunktioner beroende på i vilken position det kommer i input-strömmen (ex. monärt och binärt minus). Varje objekt som kommer efter en struktur har en vänster-prioritet. Inläsningen går så till att man parsar med en viss prioritet, och parsningen stoppar först då man hittar ett objekt med en vänster-prioritet som är mindre än den man parsar med. Parsar man alltså med en låg prioritet får man ett stort uttryck, och parsar man med en hög prioritet får man ett litet uttryck.

Vid parsningen läser varje objekt in sin del av input-strömmen. När man kommer in i konstruktionsfunktionen, så är objektet aktuellt objekt, och när man lämnar funktionen så är aktuellt objekt det objekt som förekommer direkt efter den struktur som objektet ska lämna ifrån sej.

PARSE parsar från och med nästa objekt.
XPARSE parsar från och med det aktuella objektet.
RDSIMP läser den enkla struktur som börjar med det aktuella objektet.
QPARSE testar om parsningen är färdig. QPARSE kräver att den får en struktur i sin argumentlista, och att aktuellt objekt är det objekt som följer efter strukturen.
SIMPTAB innehåller alla höger-operatorer och anadiska operatorer.
COMPTAB innehåller alla binära och vänster-operatorer.
LPTAB innehåller alla vänster-prioriteter.
De objekt som inte förekommer i tabellerna har defaultvärden, som också ligger i tabellerna.

Definition av TOP

Uttryck i REC ser TOP ut så här:

```
'top: parse 0.!
'parse: rdobj; xparse !
'xparse: qparse(x, rdsimp) !
'qparse: if getlp < x then y else
         qparse(x, rdcomp y) !
'rdsimp: eval get(simptab, obj) !
'rdcomp: get(comptab, obj) . x !
'getlp: get(lptab, obj) !
```

Här är GET en funktion som, om den inte hittar en viss nyckel i en tabell, plockar fram värdet under nyckeln DEFAULT.

I LISP ser TOP ut på detta sätt:

```
(DE TOP NIL (PARSE 0))
(DE PARSE (N) (PROG2 (RDOBJ) (XPARSE N)))
(DE XPARSE (N) (QPARSE N (RDSIMP)))
(DE QPARSE (N X) (COND ((LESSP (GETLP) N) X)
        (T (QPARSE N (RDCOMP)))))
(DE RDSIMP NIL (EVAL (GETAB 'SIMPTAB)))
(DE RDCOMP NIL (EVAL (GETAB 'COMPTAB)))
(DE GETLP NIL (GETAB 'LPTAB))
(DE GETAB (TAB) (COND ((GET TAB OBJ))
       (T (GET TAB 'DEFAULT))))
```

I detta fall är alltså X en fri variabel, som konstruktionsfunktionerna i COMPTAB använder för att referera den del som redan är inläst när man kommer till konstruktionsfunktionen.

Exempel på en grammatik

Som exempel på en grammatik anges här några exempel ur en grammatik för formel- och program-uttryck. Exemplen anges både i REC och i LISP.

Först och främst anges vad som ligger under DEFAULT i de olika tabellerna:
I SIMPTAB ligger:

```
    (rdobj; x) copy obj
resp
    (PROG1 OBJ (RDOBJ)).
```

I LPTAB ligger 21, och i COMPTAB ligger:

```
    ser(x, xparse 20)
resp
    (CONS X (REM ', (XPARSE 20))).
```

där REM definieras av:

```
    (DE REM (AT Y) (COND ((ATOM Y) (LIST Y))
           ((EQ (CAR Y) AT) (CDR Y))
           (T (LIST Y)))).
```

I LPTAB lägger man -1 under "!", vilket gör att parsningen stoppar när man träffar på ett "!", förutsatt att man har börjat parsningen med prioritet 0.

Binära och monära operatorer

Som exempel på binära och monära operatorer anges hur man skiljer mellan monärt och binärt minus. Detta klaras av om man i SIMPTAB under "-" lägger:

```
    ser('minus, parse 22)
resp
    (LIST 'MINUS (PARSE 22)).
```

I LPTAB lägger man 11, och i COMPTAB lägger man:

```
    infix(x, copy obj, parse 12)
resp
    (LIST 'DIFFERENCE X (PARSE 12)).
```

Vid monära och binära operatorer är den enda svårigheten den att välja de rätta vänster- och höger-prioriteterna, så att varje operator får så mycket man har avsett.

Nyckelordsuttryck

Som exempel på ett nyckelordsuttryck anges hur man parsar "if ... then ... else ... "- och "if ... then ... "- uttryck. Detta klaras av genom att man under "if" i SIMPTAB lägger:

```
    cond(parse 2, parse 2, if obj = 'else then parse 2 else nil)
resp
    (LIST 'COND (LIST (PARSE 2) (PARSE 2))
              (LIST T (COND ((EQ OBJ 'ELSE) (PARSE 2))
              (T NIL)))).
```

I LPTAB får man under "then" och "else" lägga 1.

Parantes-uttryck med felcheckning

För att parsa parantes-uttryck, samtidigt som man vill ha felutskrift vid icke matchande paranteser, gör man så här:

Under ")" lägger man i LPTAB 1, och i COMPTAB lägger man:

```
    prmess"icke-matchande högerparantes"; rdobj; x
resp
    (PROGN (PRIN1 '"ICKE-MATCHANDE HÖGERPARANTES")
              (TERPRI) (RDOBJ) X).
```

I SIMPTAB lägger man under "(":

```
    (if obj = rpar then rdobj; x else
            prmess" icke-matchande vänsterparantes"; x)
            parse 2
resp
    ((LAMBDA (Y) (COND ((EQ OBJ RPAR) (RDOBJ) Y)
            (T (PRIN1 '"ICKE MATCHANDE VÄNSTERPARANTES")
            (TERPRI) Y))) (PARSE 2)).
```

Här är rpar en variabel som har ")" som värde. Förklaringen till att konstruktionsfunktionerna har detta utseende är denna: ")":s funktion kommer endast att genomlöpas när man har en icke matchande ")". Ty varje "(" kommer att läsa förbi sin motsvarande ")".

Vänster-operatorer

Som exempel på en vänster-operator tas derivations-blippen. I LPTAB lägger man under "'" 17, och i COMPTAB:

```
    (rdobj; x) ser('der, x)
resp
    (PROGN (RDOBJ) (LIST 'DER X)).
```

Operator-operator felbehandling

Praktiskt taget alla konstruktionsfunktionerna måste göra någon form av inläsning, ty annars hamnar TOP i en oändlig loop. Enda undantaget från denna regel utgör felbehandlingen av två binära operatorer som står direkt efter varandra. För att klara av fallet när ett "else" följer direkt efter ";", så gör man följande: I SIMPTAB lägger man bara:

```
    nil
resp
    'NIL.
```

Effekten av detta blir samma som on det hade stått ett nil mellan ";" och "else".

Planer för 1975

Nu när TOP har visat sej vara så väldigt bra för att parsa formel- och programuttryck, så tänker vi undersöka om TOP även kan användas för att parsa naturligt språk. Det vi tänker göra är alltså ett litet behändigt programpaket, som med lätthet ska kunna pluggas in i varje program, som behöver någon form av naturligt-språk-kommunikation mellan programmet och användaren. Parsern ska klara av en förenklad form av naturligt språk, speciellt ska betydelsen av ett ord enbart bero på de föregående orden, så att ingen backup ska behöva förekomma. De problem vi ska börja med är först att undersöka vad som är en "enkel struktur" i en naturligt-språk-mening, och till att börja med ska vi göra en TOP-parser av nyckel-ords-typ som ska kunna klara av alla CICERON-meningarna.

11. PROGRAMREDSKAP & METODER PÅ NATURLIGT-SPRÅK-NIVÅ

Programredskap under denna rubrik förväntas kunna användas dels i renodlade forskningsprojekt på naturligt-språk-"förståelse", (se kapitel 15), dels som hjälpmedel för olika tillämpningsprojekt. Bl a användes nätverksparsern (avsnitt 11a) i frågesystemet för en ekologisk databas (avsnitt 13a).

11a. Nätverksparsern

Nätverksparsern har beskrivits i tidigare årsrapporter, varför vi här bara nämner det som gjorts under 1974. Under året har arbetet med nätverksparsern mest legat på en modifiering av den SEPAC-kompatibla grammatiken till en grammatik för patienceläggningssystemet (15a). Under året färdigställdes också en dokumentation av nätverkskompilatorn och runtime-systemet (1).

Vi har under året också fått hit "månstensprogrammet" (2), som användes som interface för forskare med intresse av resulteten från analysen av de prover som togs på månytan. I det systemet ingår Woods' senaste version av nätverksparsern.

Anledningen till att vi tagit hit systemet är att vi funderar på att göra en jämförande tidsstudie mellan Woods' system som är interpretativt, och vårt som kompilerar nätverksgrammatiken. En svårighet att få en rättvis jämförelse är dock att vårt system endast innehåller en delmängd av det Woodska. Vi måste alltså först göra en preliminärundersökning av om det över huvud taget är möjligt att jämföra systemen.

Referenser

1. Cedvall, M: Dokumentation av en kompilator och tillhörande runtime-system för nätverksgrammatiken, DLU 74/14, 1974.

2. Woods et al. The lunar sciences natural language information system. BBN Report no 2378, Bolt Beranek and Newman Inc, Cambridge, Mass. 1972.

A N V Ä D N I N G A V S M Å D A T A B A S E R

12. BAKGRUND OCH ÖVERSIKT

Vid sidan av arbete på grundforskningsnivå och metodutveckling har vid Datalogilaboratoriet under 1974 gjorts en intensifierad insats på tillämpningar inom smådatabas-området. Hit räknar vi då förutom problem förknippade med strukturering och hantering av komplexa datamängder, även metoder och program för kommunikation med databasorienterade system. Vårt intresse här grundar sig dels på att frågespråks- eller konversations-problemet ofta kan lösas eller förenklas med användande av små-databas-metoder, dels på att våra erfarenheter av interaktiv bearbetning av smådatabaser i stor utsträckning bör vara tillämpliga på motsvarande problem i samband med exempelvis traditionella databaser.

I tidigare årsrapporter har vi betonat vikten av samspel mellan program-redskaps-utveckling och konkreta tillämpnings-projekt. En viktig förutsättning för större satsningar är dock naturligtvis programmerings-system som kan klara program av betydande storlek, samt färdiga program-moduler för användning som standard-komponenter i olika tillämpningar. Förhållandena på dessa båda områden har under de senaste åren betydligt förbättras vid DLU, bl a genom att vårt INTERLISP-system för IBM 370 nu är i fullt bruk och vårt program-bibliotek har konverterats till INTERLISP. Ett LISP-system med betydligt mera begränsande resurser finns på tidsdelnings-datorn Simon vid Stockholms datacentral, men interaktiviteten tillsammans med vårt program-bibliotek gör även detta system användbart för tillämpnings-experiment.

Vi kan särskilja tre olika typer av tillämpningar inom smådatabas-området, nämligen:

- externa, dit vi räknar även tillämpningar som formulerats inom gruppen, men som innehållsmässigt saknar anknytning till övrig forskning.

- interna inom artificiell-intelligens-forskningen.
- interna inom annan forskning.

Inriktningen inom de interna, forskningsinriktade tillämpningarna har inte förändrats nämvärt under det gångna året. Inom artificiell-intelligens-området koncentreras fortfarande ansträngningarna på naturligt-språk-förstående system, se kapitel 15 och (vad gäller programredskap) även kapitel 11. Bland användningar i annan forskning dominerar ATMAN-projektet som beskrivits ingående i tidigare årsrapporter och som under 1974 resulterat i en doktorsavhandling. Simuleringssystemet i ATMAN behandlas i kapitel 14.

Under det senaste året är det framför allt verksamheten med externa tillämpningar som har expanderat. Kapitel 13 innehåller en redogörelse för dessa. I förra årsrapporten (DLU 1974, sid 109-119) presenterades utförligt ett antal problem-områden som är principiellt intressanta för oss och som anknyter till smådatabas-forskningen. Inom flera av dessa områden har under -74 insatser gjorts, dels inom DLU, dels i externa samarbetsprojekt och dels i form av konsultativt stöd till arbete som huvudsakligen utförts på annat håll.

I flera av dessa sammanhang har problem förknippade med interaktiv databehandling spelat en framträdande roll. Vi tycker oss ha förmärkt ett starkt växande intresse på olika håll att "förmänskliga" kontakten med ADB-system. Detta manifesterar sig exempelvis i en önskan att göra datalagrad information mer lättillgänglig genom att ersätta procedur- och batch-orienterade åtkomstmetoder med en möjlighet att via en terminal i bekväm konversation med systemet beskriva sitt informationsbehov. Ett annat exemple är det ökande intresset för att interaktivt kunna styra och studera simulerings-experiment som utförs med dator.

Under våren 74 slutfördes arbetet med den tidigare dominerande externa tillämpningen vid DLU, det frågesystem för en ekologisk databas som utförs med stöd från IBM och ingående beskrivits i föregående årsrapport sid 125 ff. Projektet sammanfattas i avsnitt 13a.

DLU har under 1974 inlett samarbete med avdelningen för databehandlingsmetoder vid statistiska centralbyrån. Inom SCB har man dels problem med att administrera och dokumentera sina ytterst omfattande samlingar av statistisk information, dels en önskan att göra denna datapotential mer tillgänglig för olika typer av avnämare. Från såväl DLU:s som SCB:s sida har här konstaterats ett gemensamt intresse-område, där ett flertal för oss intressanta tillämpningar kan urskiljas. Inom ramen för detta samarbete har ett programpaket, IDECS, utvecklats vid DLU och använts i tillämpat arbete vid SCB. Programmet, som beskrivs i avsnitt 13c, syftar till att underlätta utformningen av konversativa databas- och informationssystem och förbättra möjligheterna att låta användarsynpunkter påverka systemarbetet. I avsnitt 13b ges en kort sammanfattning av erfarenheterna med ett annat program, VARKAT-DLU, som skrevs som en förstudie till DLU-SCB-samarbetet.

Ett annat mer omfattande samarbetsprojekt som startat under året utförs tillsammans med FOA och innebär pilotprogrammering av ett simuleringssystem, där stark vikt läggs vid användarens möjligheter att interaktivt studera och påverka simuleringen; se avsnitt 13 d,e. Ett annat exemple på tillämpning av pilotsystem-typ är den förstudie som gjorts för SJ:s räkning avseende kontroll av stansunderlag för tågtidtabeller, avsnitt 13g. Vidare återfinns i kapitel 13 några mindre tillämpningar, som karakteriseras av interaktiv uppbyggnad av smådatabaser via promptningsteknik.

Ytterligare ett tillämpnings-projektet startade vid DLU under slutet av 1974, nämligen LISP-IMS som syftar till att dokumentera IMS-databaser och generera program för enklare utsökningar i dessa; se avsnitt 13h.

13. TILLÄMPNINGAR & EXPERIMENT

13a. Frågesystem för en ekologisk databas

Under våren 74 avslutades det arbete med ett frågesystem för en ekologisk databas som beskrevs i förra årsrapporten, sid 125 ff, Vi ska här kort sammanfatta projektet och resultaten därifrån.

Utgångspunkten för vårt arbete var ett tidigare forskningsinriktat samarbetsprojekt mellan IBM Svenska AB och Naturvårdsverket, där IBM utvecklade ett experimentellt relationsdatabassystem IS/1, som användes av biologer vid SNV för att göra komplexa analyser av primärdata rörande främst fiskmigration. Den ackumulerade 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 databas-bearbetningar mestadels innebärande statistisk analys eller presentation av data i form av tabeller, plottningar etc. Varje specifik frågeställning krävde i allmänhet icke-trivial programmering i IS/1:s frågespråk. Denna utfördes av data-experterna efter direktiv från SNV:s biologer.

Vår målsättning vid DLU var att utgående från de använder-formulerade frågeställningarna och motsvarande IS/1-program från IBM-SNV-projektet, implementera ett LISP-program för att förenkla kommunikationen mellan ämnesområdesexperten och hans databas. Därigenom ville vi försöka eliminera behovet av en programmerare och automatisera överföringen av en fråga till ett databas-bearbetande program.

Tack vare välvilligt tillmötesgående från IBM och SNV ställdes den tidigare använda databasen och fråge-dokumentationerna till vårt förfogande. Från början hoppades vi kunna samköra vårt LISP-program och databassystemet, men av olika skäl fick vi lov att köra IS/1-systemet på IBM:s datacentral. Detta innebar tyvärr att möjligheterna att från LISP direkt anropa databas-manipulerande rutiner fick avskrivas. I stället fick vårt program generera fullständiga IS/1-program som sedan kunde testas genom att manuellt överföras till IS/1-systemet.

Nackdelenna med ovan nämnde begränsning var framför allt den väsentligt ökade komplexiteten i problemet att generera program i ett frågespråk,

som i första hand designat för interaktiv användning vid terminal. För att förenkla testerna av genererade program, skrevs som en första del-uppgift, ett LISP-program för hantering av en relationsdatabas med IS/1:s konventioner och frågespråk. Detta var en jämförelsevis enkel uppgift och visar tydligt användbarheten av LISP för pilot-programmering eller, som i vårt fall, simulering av logiken i ett system.

En viktig målsättning i vårt arbete var även att inkorporera färdiga program-redskap ur DLU-biblioteket för att få praktiska erfarenheter av deras användbarhet i en realistisk tillämpning, såväl som av de problem som uppstår när ett antal sådana moduler kopplas ihop i ett program. Detta innebär dock ökade utrymmeskrav jämfört med ett helt specialskrivet program, i varje fall om programredskapet inte är av typ program-generator.

Vårt projekt sammanföll i tiden med övergången från LISP F1 till INTER-LISP. Detta eliminerade nackdelen med utrymmeskrävande programkoppling eftersom INTERLISP har virtuellt minne och de delar av ett biblioteks-program som inte utnyttjas i en viss tillämpning i princip inte belastar systemet.Mensamtidigt innebar det också, förutom att den första versionen av vårt program fick konverteras från LISP F1-dialekten till INTERLISP, att vissa program-redskap inte blev tillgängliga förrän ganska sent under vårt arbete.

De program-redskap ur DLU-biblioteket som har kommit till användning har varit nätverksparser och nätverkskompilator samt IKP-parser för frågespråks-analys, GIP-GUP för presentation av den små-databas med meta-information som ingår i vårt program och dessutom REMREC för modifiering av en modul så att den skulle klara de speciella kraven i vår applikation.

Vidare har TOP-parsern studerats som ett alternativ till IKP. Vår bedömning där är att TOP i princip är att föredra framför IKP vid operator- eller nyckelords-baserade frågespråk.

Den konkreta uppgiften för vårt frågesystem var att från ett frågespråk som nära ansluter sig till de ursprungliga formuleringarna kompilera databas-manipulerande program. Uppgiften sönderfaller logiskt i två

skilda delar, nämligen översättning från extern frågenotation till någon kanonisk intern representation samt generering av IS/1-program utgående från denna interna representation.

Dessutom tillkommer hantering av den meta-information om stora databasen och ämnes-området som vi behöver för såväl frågespråksanalys som kod-generering, och som vi föredrar att betrakta som en liten databas.

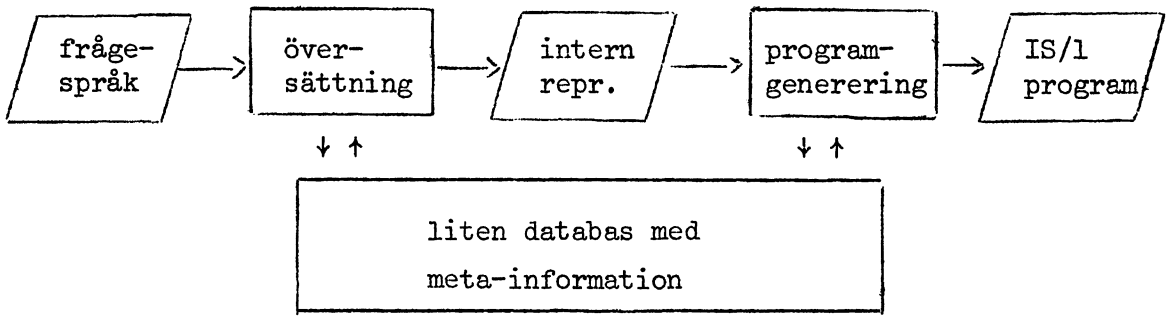


Fig 13.1 Logisk struktur

De grundläggande operationer på databasen som vi understöder i vårt system är presentationer i form av listning eller plottning samt skapande av nya datasammanställningar internt i databasen, vilka sedan kan användas för vidare bearbetning. För varje sådan operation kan ett antal parametrar specificeras, exempelvis vilka data som ska tas med i en tabell, villkor för dess medtagande samt aritmetiska eller statistiska transformationer av dessa data. Den ovan nämnda interna representationen av en frågeställning kan anses bestå av specifikation av en operation och parametrar till denna.

Generering av IS/1-program från den interna fråge-representationen sker i en specialiserad rutin för varje typ av operation. Dessa rutiner i sin tur anropar standard-funktioner för generering av kommando-sekvenser för exempelvis skapande av temporära datamängder, utförande av aritmetiska operationer etc. På grund av IS/1-språkets karaktär av interaktivt stack-manipulerande kommandospråk uppstår en del problem som gör det mindre lämpligt som generellt målspråk för programgenerering. Detta tillsammans med vår ofullständiga detaljkunskap om IS/1 fick oss att inskränka våra ambitioner till en begränsad men representativ subklass av tänkbara databas-bearbetningar. Vi utgick därvid från SNV:s material och koncentrerade oss i samband med programgenereringen primärt på de principiella problemen.

Innan vi går in på utformningen av frågespråk, ska vi kort rekapitulera förutsättningarna för den aktuella databastillämpningen. Vi har här utgått ifrån att vi i databasen har lagrat tidsserier av mätvärden, där ett enstaka värde i allmänhet har ett begränsat intresse. Användaren vill vanligtvis göra översiktliga sammanställningar eller studera aggregerade och transformerade data för att isolera statistiskt signifikanta egenskaper i materialet. Karakteristiskt för denna typ av tillämpning är lågfrekventa men tunga bearbetningar. Databasen kan ur vår synpunkt betraktas som statisk.

I den första implementationen av vår frågeinterpretator, som utfördes med hjälp av IKP-parsern, satsade vi på en nyckelordsstyrd syntax. Varje kommando åtföljdes av en argumentlista, där varje parameter föregicks av ett nyckelord. Dessa nyckelord valdes i första hand för att underlätta memorering, eftersom en databas av ovanstående typ kan förväntas användas av icke-datainriktade personer och med relativt långa mellanrum.

Den valda syntaxen fick ett på ytan naturligt-språk-liknande utseende. Detta innebär emellertid att översättaren också bör klara ett antal alternativa formuleringar av en fråga och kanske också ett visst "brus", för att den ska vara någorlunda smidig att använda. En alternativ grammatik utvecklades därför och implementerades med hjälp av nätverksparserpaketet. Denna fick dock anropa IKP för infix-till-prefix transformering av operatoruttryck.

Några typiska kommandon i vårt frågespråk är:

LIST	för tabulering av data.
PLOT	för plottning, dvs grafisk presentation.
CREATE	för skapande av en ny bestående datamängd i databasen.

Information som ska anges i parameterform är exempelvis för kommandot LIST:

- vilka värden ska definiera kolumner i vår tabell?
- vilka datamängder ska vi hämta dessa värden ifrån?
- hur ska värden från olika datamängder kopplas ihop i rader?
- ska vi ha flera tabeller för olika värden på någon variabel?
- vilka selektionsvillkor ska gälla vid framtagning av värden?

Datamängder i databasen har unika namn och dess konstituerande delmängder refereras antingen med lokala namn eller med ordningsnummer. I vårt frågespråk behöver datamängdsnamn bara anges om tvetydighet annars uppstår. Vidare sköter översättaren om att bryta ner selektionsvillkor till elementära villkor som ska gälla för en viss datamängd. För att underlätta memorering och användningen av en kommandosats som frågedokumentation finns även en facilitet för att definiera synonymer eller omskrivningar för datamängdsnamn eller satsfragment.

Vi har i vårt system i ganska ringa utsträckning utnyttjat nätverksparserns faciliteter för ordklass- och morfologisk analys. I stället har vi infört en preprocessor, vars uppgift är att normalisera en inmatad sats och exempelvis substituera interna namn i stället för användarangivna.

För att klara frågeöversättning och programgenerering används en liten databas med bl a meta-information om den ekologiska databasens innehåll. Här finns index över ingående datamängder, deras komponenter, värdeförråd m m. Vidare finns här administration av tidigare gjorda bearbetningar som sparas i ett slags programbibliotek. Detta medför att användaren med ett kommando EXECUTE kan upprepa en tidigare fråga med något modifierade parametrar, något som vi har kunnat konstatera ett klart behov av. Något mera fristående, men i princip hörande till lilla databasen, är de substitutionsregler i form av trädstrukturer, som används vid normalisering av en inmatad sats.

Erfarenheter och slutsatser. Allmänt kan sägas att arbetet inom projektet har bedrivits mer på bredden än på djupet. Vi har velat studera och implementera alla nödvändiga sub-moduler och integreringen av dessa till ett totalt system. Detta har gjort att vi inom detta projekts ram har fått ge avkall på en del motiverade önskemål om fördjupning och komplett-het. Vi anser oss dock ha visat en framkomlig väg att implementera ett system som klarar att producera databas-bearbetande program från ett användar-orienterat frågespråk. Vidare har vi ackumulerat en betydande fond av erfarenheter och uppslag som kommer att föras vidare i nu pågående projekt.

Ett viktigt resultat av projektet är även den test av DLU:s program-bibliotek som arbetet inneburit. Vår allmänna bedömning är att program-biblioteket visat sig mycket användbart för att medge koncentration på de för ett visst projekt specifika problemen. Vissa mindre problem beroende på LISP-funktioners och variablers globala karaktär har upp-stått men varit lätt avhjälpta. En annan erfarenhet är att man ofta vill göra vissa smärre modifikationer av standard-programmen. Detta visar på vikten av att dessa görs lätt modifierbara och programmeras "rent".

Av de använda biblioteksprogrammen visade sig IKP ha vissa svagheter i ovannämnda avseende. Det är nämligen en s k destruktiv parser, vilket innebär att den successivt modifierar den sats som analyseras. Detta är effektivt ur minnes-synpunkt men innebär också att program-flödet är svårt att överblicka. IKP visade sig delvis av denna anledning vara svår att använda för ett lite friare frågespråk.

Nätverksparsern å andra sidan karakteriseras av att vara nästan för kraftfull för vår tillämpning. Den är i första hand avsedd för naturligt språk, och vissa enklare konstruktioner i vårt frågespråk blev något omständliga att implementera. Till denna parsers stora förtjänster hör den rena strukturen på grammatiken och dess goda understöd för editering. Våra erfarenheter tyder på behovet att komplettera nätverksparsern med vissa faciliteter för att understödja analys av semi-formella frågespråk.

Referenser

Notley: The Peeterlee IS/1 system, IBM UKSC-0018.

Ett försök med databasteknik i miljövårdsforskning. Rapport SNV/IBN.

```
(ISX)
1. LIST TEMP , WIND IN HBO-70 OCH TOTAL CATCH IN MIGFISH FOR EACH DAY AND
   MIGRATION DIRECTION WHILE TYPE OF FISH = PEARCH :
   PARS:
2. SENT = (ISLIST TEMP , WIND IN HBO-70 OCH GROUP11 IN MIGFISH FOR
   EACH DATE AND IN-OUT WHILE SPECIES = 1)
3. PARS = (SPLITTER NIL (MIGFISH HBO-70) (TEMP (HBO-70 WIND) (MIGFISH
   GROUP11)) (DATE) (= SPECIES 1) (IN-OUT 0 1) ISLIST)

   END ;


4. IS/1 PROGRAM FOR QUERY Q-8 ;
   ============================

   DROP ;
   LOAD NULL ;
   LOAD (MIGFISH) ;
   SELECT (TOP (5) = 1) ;
   REORDER (1 , 17) ;
   LOAD (HBO-70) ;
   REORDER (1 , 3 , 2) ;
   ATTACH (1 , 1) SEP ;
   LOAD LITERAL (0) ;
   CONCATENATE ;
   REORDER (1 , 2 , 3 , 4) ;
   LOAD RELATIVE (-1) ;
   LOAD LITERAL (0) ;
   CONCATENATE ;
   LOAD LITERAL (0) ;
   CONCATENATE ;
   REORDER (1 , 3 , 4 , 2) ;
   LOAD RELATOVE (-3) ;
   REORDER (1 , 4 , 5 , 2) ;
   UNITE ;
   UNITE ;
   STORE RELATOVE (2) ;
   DROP (1) ;
   SPLIT ON (2) FROM (0) TO (1) ;
   BEGIN-IS1 ;
   LOOP :
   LOAD RELATOVE (2) ;
   LOAD LITERAL (1) ;
   SUB ;
   ON-ERROR OUT ;
   IF (TOP (1) LT 0) OUT ;
   STORE RELATOVE (2) ;
   LIST (2 , 3 , 4) ;
   DROP (1) ;
   GOTO LOOP ;
   OUT :
   END ;
   XEQ ;
   END ;
   YES ;
```

Fig 13.2 Exempel på fråga till den ekologiska databasen

1. Användarens inmatade fråga.
2. Av preprocessorn normaliserad fråga.
3. Intern representation av frågan.
4. Genererat IS/1-program.

13.b VARKAT-DLU. Ett LISP-program för simulering av ett interaktivt databassystem vid SCB

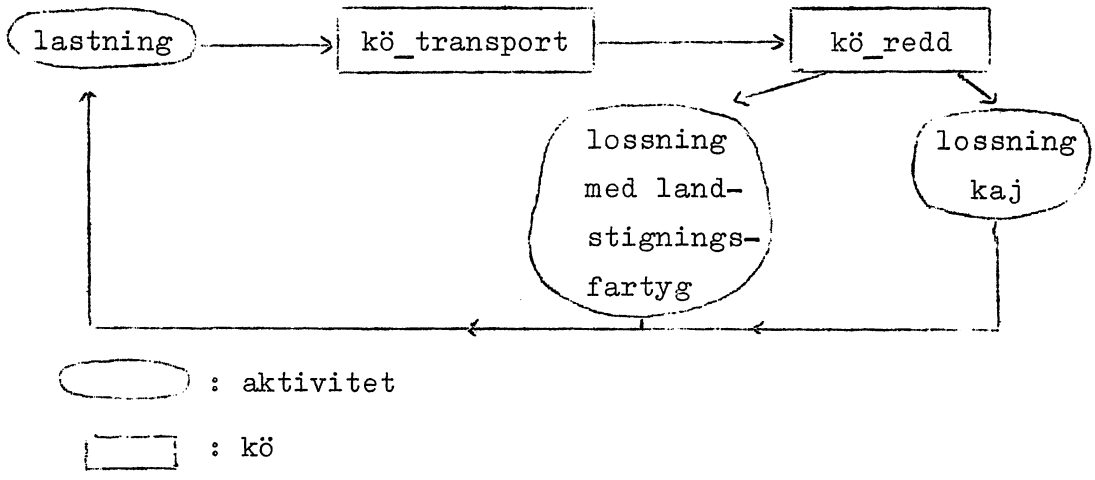
Programmet skrevs våren 74 som ett kombinerat övnings- och testprogram i samband med planeringen av sammarbetsprojektet mellan SCB och DLU. Målsättningen var dels att få en djupare förståelse för ett av SCB presenterat system, den automatiserade variabelkatalogen, dels att få testa vissa idéer i samband med pilotsystem skrivna i LISP.

SCB:s variabelkatalogsystem ger en användare interaktiv tillgång till dokumentation om data registrerade vid SCB. I dialog med systemet specificerar han vilka datamängder han vill studera och vilken information om dessa som han vill ha. Detta sker genom att med sökvillkor avgränsa en relevant informationsmängd och sedan ange de informationsslag som ska matas ut.

Vårt program VARKAT-DLU har målsättningen att upprätthålla en liten databas med variabelkataloginformation och klara interaktiv utsökning av ovanstående typ. En konversationsmonitor tar hand om användarens inmatning, analyserar sökvillkor och utmatningsspecifikation samt genererar ett utsökningsprogram. Detta program kan sedan exekveras, varvid den begärda informationen hämtas fram och skrivs ut. Vi har alltså bedömt det som mer intressant att kompilera ett specialiserat retrievalprogram i stället för att ha en parameterstyrd generell utsökningsfunktion. Från ett skelett bygger vi upp en funktion som är specialiserad och optimerad för att hämta och presentera den aktuella information.

Vid uttestningar av VARKAT-DLU visade det sig snart önskvärt att ha konversationsstrukturen flexibel och lätt modifierbar. Många aspekter på detaljutformningen var svåra att förutse på specifikationsstadiet och föranledde frekventa editeringar i programmet. Exempel på detta är svårigheter föranledda av bildskärmens begränsade format. Ofta vill man ha resultatet från flera konsekutiva interaktioner samtidigt på skärmen. Om konversationsstrukturen inte är rent hierarkisk kan det vara svårt att förutse när dessa problem kommer att uppstå. Likartade situationer uppträder när en fråga från systemet till användaren misstolkas på grund av sin kontext, trots att den sedd isolerat kan förefalla helt entydig.

Grov skiss:



Alla objekt är "räknade" (se SLISP), i och med detta kan man referera till varje objekt, som deltar i simuleringen, och bilda mängder av objekt etc.

Varje objekt står vid varje tillfälle i någon kö. En del av dessa köer är "dummyköer", dvs de är inte uppsamlingslägen, utan är bara till för att man ska kunna se vilka objekt som är sysselsatta med en viss aktivitet. Varje kö motsvarar nämligen ett händelseskede i simuleringen.

Vid ett antal ställen testar programmet på villkor, för att se om standardprocedurer ska tillämpas, eller om användaren ska få kontrollen och kunna styra simuleringen med kommandon. Användaren kan sätta dessa villkor, och alltså själv avgöra när han vill interagera.

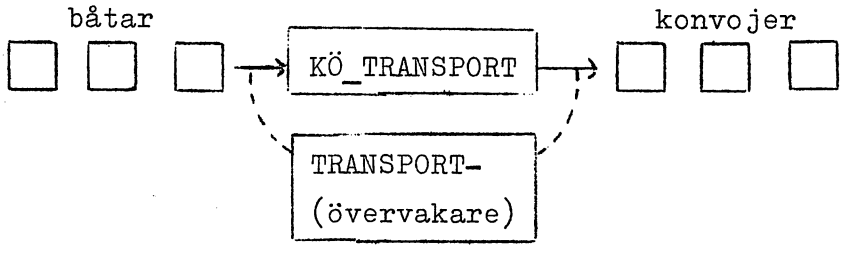
Programmet är SIMULA-tillvänt på det sättet att jag har använt SIMULA-strukturer (objekt), även om det i vissa situationer skulle ha varit bekvämare att använda LISP-listor.

Detta ger följande fördelar:

lättare att konvertera till SIMULA
lättare för en SIMULA-kunnig att förstå programmet
ger möjligheter att kontruera ett generellare kommandospråk.
ger möjlighet att använda SLISP:s SIMPRINT för objekt. (bra särskilt vid debugging).

Av sekretesskäl kan vi inte redogöra för hela programmet, men en liten sekvens av det kan belysa hur man använder köer, aktiverar objekt m m med SLISP.

Figuren föreställer ett uppsamlingsläge där fartyg har lastat färdigt och väntar i kön KÖ_TRANSPORT tills de blir tillräckligt många för en konvoj.



Kod för fartyget:

```
(INTO KÖ_TRANSPORT (THIS))          ställ mig (fartyget) i kö.
(ACTIVATEAFTER TRANSPORT (THIS))    aktivera transport-övervakaren
(PASSIVATE FTG_LOSSNING)            vila (fartyget aktiveras nästa
                                    gång vid lossning)
```

Kod för transport-övervakaren:

```
(COND
 ((LESSP (CARDINAL KÖ_TRANSPORT) KONVOJSTORLEK))          (1)
 (T (ACTIVATEAFTER (NEW KONVOJ                            (2)
                        (QELEMENTS KÖ_TRANSPORT))         (3)
                   (THIS))
    (MAPC 'PUTINTOTRP                                     (4)
          (QELEMENTS KÖ_TRANSPORT))))
(PASSIVATE TRANSPORT_START)                               (5)
```

1) inte tillräckligt många båtar för en konvoj.
2) en ny konvoj skapas och aktiveras.
3) som dynamiskt attribut till konvoj sätts en lista på de fartyg, som är med i konvojen.
4) ställ fartygen i kön TRP, som visar att de är under transport.
5) vila tills nästa fartyg aktiverar mig.

Kod för konvojen:

```
(HOLD KONVOJ_E (TPTID BATAR TP_STRÄCKA))     vänta båtarnas trans
                        ↑                    port tid.
                    dynamiskt
                    attribut för konvoj.
```

Kommandospråk

2 olika (ytterlighets) filosofier:

1) operatören kan inte styra förloppet, utan bara observera hur simuleringen fortlöper.
2) operatören kan ändra på samtliga variabler som styr programmet och han har själv ansvaret för att hans ändringar inte förstör körningen.

Ingen av dessa två varianter är önskvärd. Helst vill man ju behålla både den fulla säkerheten och tillgång till det som styr simuleringen. Som vårt kommandospråk nu är kan man styra allt, som vi har tänkt att man skulle vilja styra, och säkerhetskontroller görs så långt som möjligt.

Användaren kan under simuleringens gång ge kommandon för att:

1) införa nya objekt i simuleringen
2) ta bort objekt
3) bilda och namnge mängdbeskrivningar och mängder av objekt
4) titta på objekts dynamiska och statiska attribut
5) ändra på objekts attribut
6) få ut tabeller på attributsummor och medelvärden för olika mängder av objekt
7) titta var i simuleringen som ett visst objekt eller en mängd av objekt befinner sig
8) titta vilka objekt som finns i ett visst skede av simuleringen.
9) spara simuleringen för att senare gå tillbaka till samma punkt och styra simuleringen på ett annat sätt.
10) titta och ändra på vissa variabler i programmet.
11) få ut tabeller på "tillväxt" från simuleringens start tills nu.
12) sätta önskad grad av interaktivitet.
13) köra simuleringen ett visst antal timmar framåt, eller till dess att ett visst villkor (som ges av operatören) är uppfyllt.
14) lista alla medlemmar i en kö.

Här följer en dialog med programmet:

```
*STARTA 3 ST LFTG;                          LFTG1 är en typ av fartyg.
 (LASTNINGSHAMN ?) * NIL;                   får gå till valfri hamn
 (LOSSNINGSHAMN ?) * NIL;
* VISA LASTNING;
  QUEUE LASTNING
        MEMBERS=
        (LFTG1 1)
        (LFTG1 2)
        (LFTG1 3)
* STARTA 10 ST HFTG1 KL 2;
 (LASTNINGSHAMN ?) * IHAMN:2;
 (LOSSNINGSHAMN ?) * UHAMN:1;
*M1 := LASTNING;                            M1 är en mängdbeskrivning
* M2 := HFTG1 SOM STÅR I LASTNING;
* S1 := SPAR M1;                            S1 är en (fix) mängd
* VISA M1;
  ((LFTG1 1) (LFTG1 2) (LFTG1 3))
* VISA M2;
* VISA M2;
  NIL                                       HFTG1 är ännu inte startade
* VISA S1;
  ((LFTG1 1) (LFTG1 2) (LFTG1 3))
* VISA TIDSKÖN;
 THE SYSTEMS TIMEQUEUE=
0   *MAINPROGRAM
1   (LFTG1 1)
2   (LFTG1 2)                 lastar (färdiga kl 1)
1   (LFTG1 3)
2   (HFTG1 1)
 ...                          väntar på att få starta
2   (HFTG1 10)
* KÖR 1 TIMME;^
 KLOCKAN ÄR NU 1
*VISA TIDSKÖN;
 1  *MAINPROGRAM
 2  (HFTG1 1)
```

```
 ...
 6  (KONVOJ 1)
 6  (KONVOJ 2)          3 st LFTG1 har nu lastat färdigt och ingår
 6  (KONVOJ 3)          i konvojer.
* VISA KONVOJSTORLEK;
 1
* KONVOJSTORLEK:=2;       vissa variabler går att ändra.
 2                        nu kommer konvojerna att innehålla
* KÖR 2 TIMMAR;           2 fartyg.
 KLOCKAN ÄR NU 3
* VISA M1;                har nu förändrats
 ((HFTG1 1) (HFTG1 2) (HFTG1 3) (HFTG1 4) (HFTG1 5))
* VISA S1;                är lika som förut
 ((LFTG1 1) (LFTG1 2) (LFTG1 3))
* VISA KÖER;
                    HFTG1 .. LFTG1  ...
KÖ_HAMN_LASTNING      5   -    -
LASTNING              5   -    -
TRANSPORT             -   -    3
* KÖR 60 TIMMAR;
 KLOCKAN ÄR NU 63
* TILLVÄXT TOTALT;
 EFTER        TOTTRUPP  UH          TRUPP1     TRUPP2
20 TIMMAR     0.51       0          0.51       0
40 TIMMAR     0.51       0          0.51       0
60 TIMMAR     0.69       0          0.69       0
```

13e <u>LME kopplingsscheman</u>

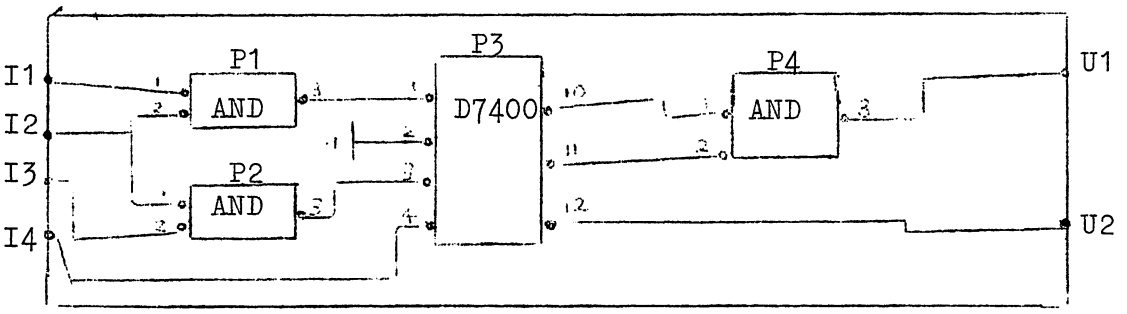
Under våren 1974 implementerades LISP F2 hos LM Ericsson AB. Som demonstration av systemets användning inom LM Ericsson skrevs programmet FAIRT. FAIRT är avsett att

a) interaktivt mata in kopplingsscheman för kretskort <u>samt</u>
b) utföra vissa tester om beräkningar ur det givna kopplingsschemat.

I inmatningsfasen utnyttjas en promptningsteknik där programmet hela tiden håller reda på vilka anslutningar som skall ges av användaren. Det är alltså omöjligt att "glömma" någon anslutning (även

om man naturligtvis kan råka koppla anslutningar till fel komponenter)

Ett exempel: Givet följande kopplingsschema:



Komponenterna döps till namn som börjar med P. (Här P1, P2, P3, P4).

Komponenttypen anges inuti komponenter på schemat (Här AND och D7400)

Typen är antingen en enkel logisk komponent (And, OR etc) eller en sk kapsel innehållande ett antal enkla komponenter (här D7400).

För varje kapsel är det definierat vilka stiftnr som är ingångar och för vilka som är utgångar. (Denna information finns förlagrat i FAIRT programmet).

För enkla komponenter har schemakonstruktörenfrihet att själv välja dessa stiftnr. Dessutom finns ingångar (börjar med I) utgångar (börjar med U) Jord (J) och Plus (P).

Schemakonstruktören sätter sig vid terminalen, kallar in FAIRT programmet och exekverar funktionen (BUILD) varpå följande dialog utspinner sig. (Prompting från FAIRTEST är understruket).

GE EN UTGÅNG*U1
DRIVS AV?*P4$3 (komponent P4
TYP P4?* AND stift 3)
INGÅNGS-BEN?*(1 2)
DRIVS AV?
1 ?*P3$10
2 ?*P3$11
TYP P3?* D7400
DRIVS AV?
1 ?* P1$3
2 ?* J
3 ?* P2$3
4 ?* I4

TYP P1?* AND
INGÅNGS-BEN?* (1 2)
DRIVS AV ?
1 ?* I1
2 ?* I2
TYP P2 ?* AND
INGÅNGS-BEN?* (1 2)
DRIVS AV?
1 ?* I2
2 ?* I3
FLER UTGÅNGAR ?* U2
DRIVS AV ?* P3 $ 12
FLER UTGÅNGAR? * NEJ
OK.

När schemat är inmatat. kan man erhålla diverse utskrifter som t ex

- alla förbindelser från-till en viss komponent.
- alla komponenter (med eller utan förbindelser)
- antal komponenter fördelade på typ
- utdata till standardprogrammer FAIRTEST
- återkopplingar i schemat
- sk virningar (två eller flera utgångar är ihopkopplade)

Åter kopplingar och virningar bör konstruktören vara medveten om. Att dessa finns (om inga oväntade finns) är en snabb test på att grafen är riktigt inmatad.

Utdata till FAIRTEST programmet är en av de viktigaste funktionerna.

FAIRTEST är ett standardprogram, som är dyrt att köra och som har komplicerade styrkort för att beskriva ett schema. När schema konstruktören kontrollerat att schemat är korrekt inmatat (vilket är lätt att kontrollera i FAIRT). skriver han

(STYRKORT)

och styrkorten till FAIRTEST läggs ut på angiven fil.

Programmet FAIRT demonstrerades på våren 74 och väckte stort intresse. Det användes även som exempel underlag vid LISP-kurs hos LME under hösten 74. Det övertogs av LME i samband med inköp av LISP F2 och utvecklades vidare. Av sekretess-skäl gentemot LME kan DLU ej idag rapportera status på programmet.

13f Kontroll av tågtider

Under höstterminen 1974 tog en utvecklingsgrupp från SJ kontakt med DLU. SJ planerar för närvarande att lägga hela järnvägsnätet + övrig information (priser etc) i en databas varur man skall kunna erhålla kommunikationstabeller, prislistor, snabbtabeller för vissa stationer, färdplan för lokförare osv. Gruppens uppgift är bl a att studera vilket språk resp databas-system som skall användas, och ville veta om INTERLISP var ett realistiskt alternativ.

För att demonstrera INTERLISP:s kapacitet som programmeringsspråk skrevs ett demonstrationsprogram för en av deluppgifterna i det stora projektet.

Indata till programmet var dels tågsträckor, dels stationsuppgifter.

Tågsträckorna bestod av ett sträcknummer, ett antal stationsnamn samt kilometerangivelser.

Stationsuppgifterna bestod i ett antal tåguppgifter för varje station För varje tåg var noterat tågnr, dag (vardag, söndag etc), ankomsttid, avgångstid, spår, kod.

Kod angav bl a om det var godståg eller persontåg.

Det material som behandlades hade hämtats från ett område i Småland och bestod av 12 tågsträckor med ca 5 stationer per sträcka samt 48 stationer med ca 20 tåg per station. (Stansunderlaget utgjorde 27 A4 sidor).

Uppgifter var nu att kontrollera materialet så långt som möjligt. Kontrollen delades upp i två typer:

Kontroller som kunde utföras oberoende av andra data.
Ex. tider är klockslag, spår är tal i (1, 11), km är tal i (0, 1000), dag kod är något av vissa koder etc.
Dessa kontroller är triviala att utföra och i fortsättningen ointressanta.

Indirekta kontroller:

Nu kontrolleras, att tågen går som de ska mellan stationerna. När hela materialet är inläst, kan man för varje tåg rekonstruera mellan vilka stationer det går. Kontroller som utföres är av typen. Tåg får ej "hoppa över" station på en sträcka.
Tåg skall ha samma dag-kod längs en sträcka.
Tåg skall färdas med rimlig hastighet mellan stationer.
Två tåg får ej stå på samma spår samtidigt
osv.

Dessa indirekta kontroller är svåra att göra i ett konventionellt språk (som saknar liststrukturer och symbolbehandling) men lätta att utföra i INTERLISP.

Hela programmet tog två veckor att skriva, testa ut och dokumenterar.

Att uppgiften löstes på så kort tid imponerade mycket på gruppen från SJ.

Fn (jan 1975) väntar DLU på initiativ från Sj för vidare ev samarbete i syfte att utröna INTERLISP:s duglighet som programmeringsspråk för de problem Sj ställs inför.

13g BNF - Generering av testdata

Under våren 74 skrevs ett program varmed man enkelt och säkert lagrar en BNF-grammatik. Programmet är ett experiment med generering av en databas via prompting.

En BNF-grammatik kan ses som en graf med en bestämd startnod. Programmet begär startnod, subnoder till startnoder, subnoder till varje nod osv. Eftersom programmet "håller reda på" vilka noder som ännu ej är specificerade är det ingen risk för att man glömmer noder.

Felstavade noder upptäcks också lätt då programmet så småningom begär in uppgifter om dessa.

Tekniken att interaktivt via promptning bygga upp en graf gav i impulser till programmet för LME:s kopplingsscheman (se kap 13e).

BNF-grammatiken har sedan använts för

- generering av testdata
- generering av "slump-program" (av elever på en kompilator teori kurs, för att kontrollera att en viss grammatik genererar förväntade slags satser).
- generering av naturligtspråk (på svenska, som demonstrations-program vid DLU, på engelska, som demonstrationsprogram på IFIP-kongressen).

Exempel på out put från BNF-programmet med engelsk grammatik inlagd:
(Taget från IFIP-demonstrationen)

```
(story)
 GOOD AFTERNOON
 NICE TO SEE YOU, AND HOW ARE YOU TO-DAY?
 I HOPE YOU DON'T MIND GIVING ME YOUR NAME
*snoopy
 WELL THEN SNOOPY, I WOULD ALSO WANT TO KNOW IF YOU ARE A MALE OR
 A FEMALE.*dog
 THAT'S FINE SNOOPY
 JUST A MOMENT, I'VE GOT OTHER THINGS TO DO FOR A LITTLE WHILE, YOU
 SEE.
 WELL I'M READY. YOU SEE, I WOULD LIKE TO TELL YOU A LITTLE STORY.
 YOU DO WANT TO HEAR IT, DON'T YOU?
*yes
 HERE IT IS:
 " DO YOU OFTEN SWEAR "? SAID SNOOPY WHEN SNOOPY CAME TO IFIP IN THE
 NIGHT. " NATURALLY! " A VOICE MURMURED. THE BEES WERE HUMMING AND
 A LONELY CAT WAS MEWING. ALICE IN WONDERLAND, WHO DOESN'T EVER GO
 ABROAD AND WORKS FOR BURROUGHS, WENT TO GREENLAND. SHE VANISHED. OLIVER
 HARDY WAS EXTREMELY SAD AND AKSED: " WHY ARE YOU SO UGLY? ". "WOULD
 YOU PLEASE REPEAT THAT ", LONG JOHN SILVER ANSWERED SLOWLY. THEN MR
 HIDE ASKED QUICKLY: " HAVE YOU SEEN BATMAN IN THE AFTERNOON? ". "
 YES! " WAS THE ANSWER. THE CLOUDS WERE LOOKING VERY MENACING AND IT
 WAS DARK AND CLOUDY. BATMAN, WHO WORKS FOR BURROUGHS AND WORKS FOR
 MINISTRY OF BETTER TIMES, WENT TO UPPER SUDAN AND WENT OFF TO GET
 A DRINK. NOT A CLOUD WAS SEEN AND THE CLOUDS WERE LOOKING VERY MENACING
 THEN HE WONDERED: DID YOU SEE THE EXORCIST. " HAVEN'T GOT A CLUE!
 " BATMAN THOUGHT. THEN HE SAID: " BATMAN SOMETIMES SNORES TIMIDLY!
 WHEN DOCTOR KILDARE UNPLUGS HIS ELECTRIC RAZOR ". HE WAS UTTERLY
 SATISFIED AND WENT TO STOCKHOLM. HE COULDN'T HELP BEING PRETTY SLEEPY
 AND ASKED: " DO YOU FEEL TIRED ". " YES! " WAS THE ANSWER. THE SKY
 THREATENED RAIN AND IT WAS DARK AND CLOUDY. THEN HE SAID: " ALICE
 IN WONDERLAND YAWNS ALOUD WHEN DOCTOR JEKYL UNPLUGS HIS ELECTRIC RAZOR
 ". A BABY WAS CRYING DESERTEDLY AND A DOG WAS BARKING SOMEWHERE. HE
 RUSHED INTO THE SHOWER. HE SAID: " MR HIDE OFTEN YAWNS ALOUD WHEN
 DOCTOR KILDARE BREAKS HIS RIGHT ARM " AND TOOK A BATH.
```

ANOTHER ONE
*yes
HERE IT IS:
IT WAS STORMING AND AN AMBULANCE PASSED BY WITH ITS SIRENS SOUNDING.
THE SUN WAS SHINING AND AN AMBULANCE PASSED BY WITH ITS SIRENS SOUNDING
THEN HE QUESTIONED: " WHERE IS BATMAN ". " THAT'S NOT FOR YOU TO
ASK! ", WAS THE ANSWER. IT WAS A SUNNY DAY. ALICE IN WONDERLAND REALLY
GOT JOLLY HAPPY AND WENT TO NEW YORK. WHEN SNOOPY ENTERED IFIP SHORTLY
AFTER BREAKFAST. AN AMBULANCE PASSED BY WITH ITS SIRENS SOUNDING AND
YOU COULD NOT HEAR A SOUND ANYWHERE. SUDDENLY STAN LAUREL QUESTIONED
SLOWLY: " WHO HAS GONE TO ROME? ". " MR HIDE ", MR HIDE ANSWERED SLOWLY
AND STARTED TO LIGHT A CIGARETTE. YOU COULD HEAR CHILDREN PLAYING
IN THE STREET. HE COULDN'T HELP BEING JOLLY ANGRY AND QUESTIONED:
" WHERE IS DOCTOR JEKYL? ". " ON THE WAGON! ", LONG JOHN SILVER THOUGHT
AND VANISHED. SUDDENLY HE ASKED POKING HIS NOSE: " HAVE YOU GOT ANY
GALOSHES AT BJORN BORG'S? ". " NATURALLY! " BATMAN THOUGHT AND WENT
HOME. HE WENT TO GREENLAND AND TOOK A BATH. THE DARKNESS CAME ON.
THE SUN WAS SETTING.

ONE MORE?
*!
?:
D

****** RESET *******

<u>Exempel på output med svensk grammatik inlaggd</u>:

TIDIGT I MORSE, NÄR DEN DOMINERANDE, BLÅÖGDA OCH RÖDKINDADE
FRIMÄRKS-MOTSTÅNDAREN KALLE MÄKILÄ, HAN MED DE UPPSEENDEVÄCKANDE
BYXORNA NI VET, SOM ÄR SÅ FANTASTISKT TRÅNGSYNT, KOM TILL
DLU, TRYCKTE HAN TÅRTFATET I HANDEN PÅ HONOM. DETTA TVINGADE
HONOM ATT KASTA DEN LYCKLIGTVIS NYTÖMDA ASKKOPPEN I HUVUDET PÅ
HONOM..

DÅ NITISKA 3-BETYGAREN ANDERS BECKMAN SOM SYNBARLIGEN HAR
EN ALLMÄNBILDAD SKRÄDDARE PÅ LABBET, STEG IN I LABBET TIDIGT
I MORSE, UNDRADE HAN VETGIRIGT: " ÄR ANDERS HARALDSSON LÅNGHÅRIG?
" NEJ TYCKTE HAN. DEN RÖDKINDADE ARBETSNARKOMANEN MATS NORDSTRÖM
BLIDDE HÄMNDLYSTEN OCH KNACKADE BREVPRESSEN AV MARMOR I FAMNEN
PÅ MATS CEDVALL.

DÅ DEN PORTVINSTÅFÖRSEDDA OCH CIGARRÖKANDE DLU-SAMLAREN STAFFAN
LÖÖF, HAN MED FINNAR PÅ NÄSAN OCH FINNAR PÅ NÄSAN, SOM LÄSER
LÅNGSAMT, STEG IN I DLU VID KAFFET, VÄLTE HAN ETT ENGELSK-KINESISKT
LEXIKON I HÅRET PÅ HONOM. DETTA KOM DENNE ATT DÄNGA DEN HALVÄTNA
SEMLAN I HUVUDET PÅ JACOB PALME'S BROR..

NÄR DEN DOMINERANDE KAMEL-ÄLSKAREN STURE HÄGGLUNG, HAN MED
DE UPPSEENDEVÄCKANDE BYXORNA, DEN SKÄRBLOMMIGA SLIPSEN, ÖLMAGEN
OCH ÖLMAGEN NI VET: KOM TILL SYSTEMET VID KAFFET, SPORDE
HAN: " ÄR GLÄDJEDÖDAREN ERIK SANDEWALL INTE NÄSTAN ORGINELL?
" NEJ REPLIKERADE DENNE. DÅ ANSÅG HAN: " ERIK SANDEWALL,
DEN SNORKIGA SPELAUTOMAT-MOTSTÅNDAREN, SLÅR PORTFÖLJEN I
FICKAN PA OLLE WILLEN! ".

IMORSE, NÄR FORTRAN-ÄLSKAREN HERR URMI KOM TILL HEMMETS LUGNA VRÅ, YTTRADE HAN: " STURE HÄGGLUND SOM SÄLLAN HAR EN KLOK VAKTHUND, ÄR GLADLYNT! ". DENNE FORTSATTE ATT VARA HÄMDLYSTEN OCH SPORDE: " HAR IBM-ENTUSIASTEN MATS NORDSTRÖM EN CHARMERANDE KOMPIS? " NEJ PÅSTOD HAN VARVID HAN STOPPADE BLOMSTERVASTEN I SKALLEN PÅ STURE HÄGGLUND, DEN LILLGAMLA ARBETSNARKOMANEN,. DÅ UNDRADE HAN: " VARFÖR HAR ANDERS HARALDSON, HAN MED DEN PIPIGA RÖSTEN ALLTSÅ, GENOMGÅENDE EN GANSKA KONSERVATIV STÄDERSKA? " JA TYCKTE HAN VARVID HAN SLOG PROGRAMLISTAN I BAKEN PÅ DENNE. HAN ÖVERGICK TILL ATT BLI OMDSINT OCH UNDRADE: " ÄR DEN EGOCENTRISKA OCH CIGARRÖKANDE GLÄDJEDÖDAREN TORGNY THOLERUS, HAN MED ÖLMAGEN NI VET, SOM ÄR CHARMERANDE, SÅ MAXIMALT FÖRSTÅNDIG? " JA REPLIKERADE HAN. DETTA FÖRANLEDDE DENNE ATT TAPPA DEN REDAN TRASIGA HÅLSLAGAREN I SKALLEN PÅ DENNE:. DÅ UNDRADE HAN: " ÄR DEN CIGARRÖKANDE, EVIGA, SJÄLVUPPOFFRANDE OCH IVRIGA NIKOTINISTEN ERIK SOM UNDANTAGSLÖST STOPPAR PORTFÖLJEN I KNÄT PÅ TORGNY THOLERUS'S SYSTER, GENOMGÅENDE GANSKA SPIRITUELL? " JA REPLIKERADE HAN VARVID DENNE PLACERADE DEN REDAN TRASIGA HÅLSLAGAREN I FAMNEN PÅ HONOM.

13h LISP-IMS MANAGER, ett system för strukturbeskrivning av IMS-databaser

Ett nytt projekt, kallat LISP-IMS MANAGER (LIM), har just (dec 1974) initierats i samarbete med Uppsala Datamaskin Central (UDAC). Det syftar till att beskriva den logiska och fysiska strukturen i en IMS (Information Management System) databas. En sådan databas-beskrivande databas skall sedan bl a användas till att generera program (i något av språken COBOL, PL/1, FORTRAN eller assembler) för retrieval från IMS-databasen. Man vill t ex kunna göra snygga utskrifter i klartext av specificerade delar av sin IMS-databas. Genom att låta LISP-IMS generera dessa program i stället för att man gör dem själv, har man lättare att reorganisera strukturen i IMS-databasen utan omfattande omprogrammering.

LIM kan vidare användas som en dokumentationsdatabas för IMS-databaser, för konsistenstester m m.

Till dags dato har utformandet av den interna LISP-strukturen för att beskriva IMS-databasens struktur studerats och specificerats. Vidare har rutiner för in- och utmatning av dessa strukturer konstruerats.

Som utgångspunkt för beskrivningen av IMS databasens struktur i LISP används de assembler makron, som brukas vid uppläggning av sk Data

Base Descriptors i IMS. Dessa "DBD-makron', som beskriver strukturen för IMS-systemet, överförs till en intern pekarstruktur i LISP. Det är lätt att skriva LISP procedurer, som automatiskt överför DBD-makrona till motsvarande LISP-struktur. För närvarande används dock en mer LISP orienterad notation. Nedan följer ett exempel som illustrerar motsvarigheten mellan DBD-makron och LISP-IMS strukturen. För beskriva LISP-atomer med tillhörande egenskaplistor används notationen

(PROPL a i_1 p_1
i_2 p_2
.....
i_k p_k)

a betecknar den atom vars egenskapslista skall beskrivas.

i_n betecknar indikatorn nr n på a:s egenskapslista

p_n betecknar det som finns lagrat på a:s egenskapslista under indikatorn i_n .

DBD makron:

```
DBD NAME = SKILLINV, ACCESS = HSAM
DATASET ...
SEGM  NAME = SKILL, BYTES = 21, FREQ = 100
FIELD NAME = TYPE, BYTES = 21, START = 1, TYPE = C
SEGN  NAME = NAME, BYTES = 20, FREQ = 500, PARENT = SKILL
....
```

Motsvarande LISP-struktur:

```
(PROPL SKILLINV NAME SKILLINV
                CLASS DATABASE
                ACCESS HSAM
                ROOT S1
                ....                )
```

```
(PROPL S1   NAME SKILL
            BYTES 21
            FREQ 100
            CLASS SEGMENT
            DATABASE SKILLINV
            CHILDREN (S2)
            FIELDS   (F1))

(PROPL F1   NAME TYPE
            CLASS FIELD
            BYTES 21
            START 1
            TYPE C
            PARENT S1)

(PROPL S2   NAME NAME
            CLASS SEGMENT
            BYTES 20
            FREQ 500
            PARENT S1
            ....)
```

14. ANVÄNDNINGAR I ANNAN FORSKNING

14a. Simuleringssystem för ATMAN

Atmanprojektet har de senaste fem åren arbetat med att utveckla ett system för simulering av kausala modeller av socialt handlande. Systemet har beskrivits i tidigare årsrapporter och finns nu också publicerat i fem rapporter (1).

Två simuleringssystem för ATMAN projektet har utvecklats. Det ena av dessa har skrivits i LISP, det andra i programmeringsspråket REC. Nedan följer utdrag, som beskriver delar av dessa simuleringssystem, ur boken "Homo Cyberneticus", vol 3. Först följer en definition av 'jig' begreppet (en form av processer), som är av central betydelse för simuleringssystemen. Därefter följer i tur och ordning simuleringssystemet skrivit i LISP resp REC.

Jigs

Our concept 'jig' is meant to allude both to "template" or "stereotype" and to "jig-saw puzzle". Our jigs describe behaviour steoreotypes, and they are also small components in a large 'picture' of social reality.

Jigs are essentially subroutines or functions. Their only peculiarity is that they can be ascribed properties and that these properties govern their evaluation. In fact, during a simulation run there will be many copies made of these jigs; these copies are called instances and each will have individual properties.

Jigs (and thus also their instances) describe activities. Jig instances are always associated with certain database entities (individuals or contexts); when an instance is evaluated, it thus describes a current activity of the entity.

For example, let JNR be an instance that describes the activity 'sleep', and let JNR be associated with an entity of type 'person', called Self. When the monitor, following its queuing order among the instances that are to be evaluated ('executed'), happens to execute JNR, we can say that Self sleeps.

The body of a jig is the sequence of functions or rules that define a possible activity of an entity. Such a sequence describes in utmost

detail what is to happen to the database if a person is to go shopping, or to enter a bus, for example.

ATMAN in LISP ('ATMAN2')

Atman2 Components

We can distinguish four relatively independent parts ('package') within Atman2:

(a) GETT/PUTT, which is a 'package' of functions of use for the storing and fetching of information in property trees (cf. ref 2)

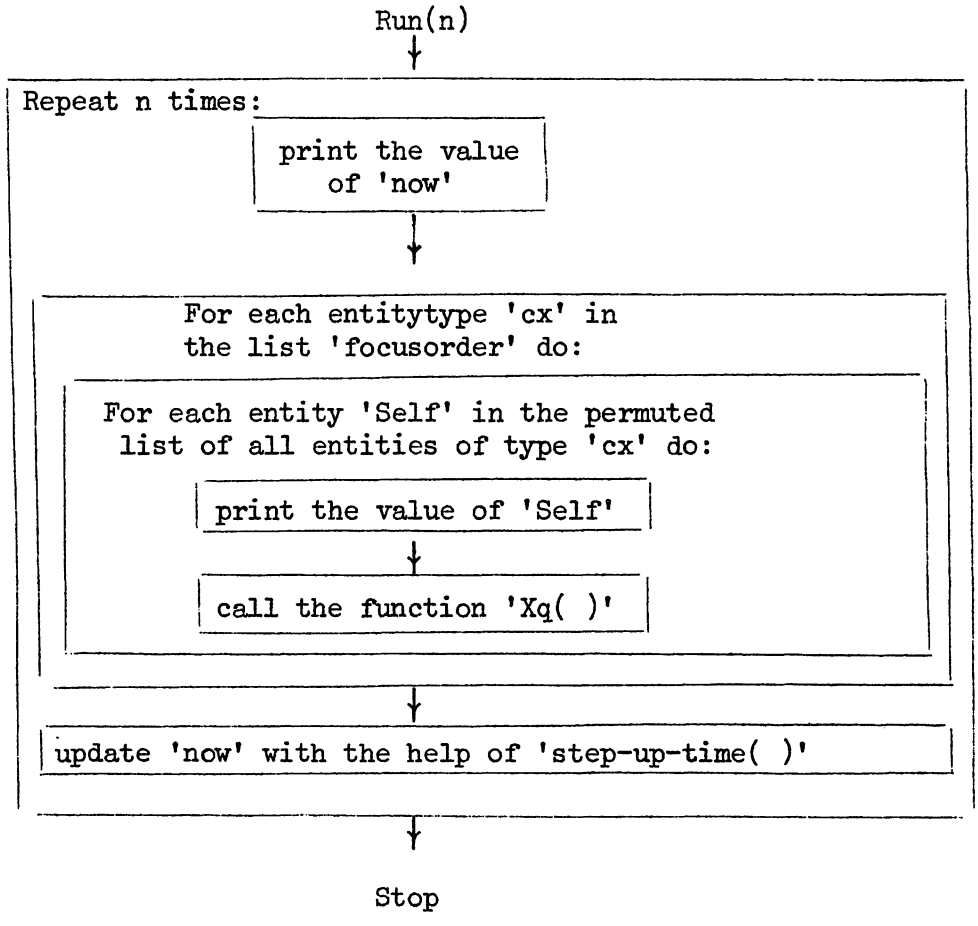
(b) RANDOM, which comprises functions for samplings of various kinds.

(c) CREATE, which comprises functions of use for generating the initial database (cf. Part +9.

(d) ATSIM, which is the set of central monitoring rules.

A package which furnishes macro-definition facilities (that is: enables one to define a private input language, tailormade for the problems one intends to formulate) has furthermore been employed, but is not a necessary program part. (cf. ref 3).

The Top Loop. The flow chart for the top loop is as follows:



"Now" denotes the present time point. 'focusorder' is a list of entity types. 'Xq' is the jig processsor in ATSIM; it is iteratively applied to all instances JNR in the jig queue of Self.

The Jig Processor

When 'Xq( )' is called, 'Self' is already bound to the presently focused entity. At the database node 'Self::Taskstack' it finds the relevant jig queue which it will work on.

This jig queue is thereafter transferred to the node 'Jigkew'. 'self::Taskstack' obtains an empty list instead. 'Self::Attention' is set to zero.

(a) The first element of 'jigkew' is bound to the variable (or node) JNR. 'jigkew' is rebound to CDR of the list at 'Jigkew'. ("CDR of list L" means "the list L except its presently first element".)

(b) The function 'Xql' is applied to JNR. The execution-exit value returned by 'Xql' (cf. Part I/7.2) governs what is to happen with JNR after this application:
If "Postpone" is returned, JNR's execution is postponed.
If "Continue" is returned, JNR's execution is continued.
If "Occupied" is returned, Self is occupied with the execution of so many instances that there is no 'attention' left for any additional jig executions. JNR and the list at 'Jigkew' are put back onto Self's task stack (where also all postponed or continued instances have been put).
If 'Xql' returns any other value than these, JNR is abandoned.

(c) If "Occupies" has not occurred, and 'Jigkew' is not empty, go to (a), else return control to the top loop.

Jig Execution

The execution of JNR is performed by the function 'Xql' when it is applied to JNR. It 'behaves' as follows:

(a) If 'JNR::Xqcontrol' is not NIL, its value is returned.

(b) If 'JNR::Duration' is less than half of the value of 'Fugit', "Abandon" is returned.

(c) If 'JNR::Succedes' is not NIL, "Postpone" is returned.

(d) If 'JNR::Xqstarted' is NIL, it is given the current value of 'Now'.

(e) If the sum 'Self::Attention'+'JNR::Attention' exceeds 150, or if 'Self::Attention' exceeds 100 while 'JNR::Attention' exceeds 1, "Occupied" is returned.

(f) Otherwise, add the value of 'JNR::Attention' to 'Self::Attention'.

(g) If 'JNR::Duration' is NIL, or if ('JNR::Xqstarted' + 'JNR::Duration') exceeds 'Now', 'Xql' returns the result of the jig body's evaluation.

(h) If 'JNR::Xqlatest' is NIL, it is set to zero. If ('JNR::Xqlatest' +'JNR::XqEvery') exceeds 'Now', "Continue" is returned without jig-body evaluation.

(i) Otherwise, 'Xql' returns "Continue" after jig-body evaluation.

Explicit execution conditions can, during the jig-body evaluation lead to direct exits from 'Xql',usually with "Postpone" as exit value. Such exists are thus not under the control of the jig processor but of the executed jig instance.

Jig Definition

A jig is defined by an expression of the form "(DEFJIG <name> <argument list> <list of jig properties>.<jig body>)"
where
<name> is the name of a LISP atom;
<argument list> specifies additions to the 'environment' in which the jig body is to be evaluated; <list of jig properties> contains explicit value assignments to the jig properties; the form is "(P1 V1 P2 V2 ... Pn Vn)" where each "P" denotes a jig-property name, and each "V" an appropriate value; the "P" are not evaluated, but the "V" are; <jig body> is a sequence of LISP functions; and the dot "." indicates that the extent of the jig body is not to be enclosed in separate parentheses; all following expressions are instead counted as part of the jig body. The dot is not written.

Example:

```
(DEFJIG GREET (X)  (ATTENTION (APPROX 20)
                              PRIORITY 8)
    (PRINT (LIST 'HELLO X)) (PRINT
                             'HOW DO YOU DO)) )
```

The jig body is automatically 'macro-expanded', that is, the submitted code is modified such that defined abbreviations and other explicit syntactic formulation conventions ('macros') are replaced by their usually more complicated (but computionally more effective) correspondences. (Cf. ref. 3).

Pushing

(Jigs can in principle be used as if they were ordinary LISP functions; the atman monitor will then not govern them and no tacit execution conditions will therefore be tested. If there are any explicit execution conditions, however, they will be part of the jig body and therefore be evaluated with it. However, in the Atman2 implementation we have not made provision for such extra-monitor use.)

To submit a jig to the control of the atman monitor, we need to push it. Pushing a jig implies that the monitor makes a copy ('JNR') of it and attributes JNR with all given jig-property values. JNR is then added to the jig queue of some entity.

Pushing is commanded by expressions of the form

"(PUSHJIGQ <J> <E>: <list of jig properties>)",

where <J>is itself an S-expression of the forms

"(<jig name> <arguments>)",

where <arguments> is zero, one, or several S-expressions, or

"(APPLY:: <name identification> <arguments>),

where <arguments> has the above meaning, and <name identification> is to evaluate to a jig name;
<E> is the name of a database entity; if <E> evaluates to NIL, its default

value is the current value of 'Self'; and the dot "." and <list of jig properties> have the same meaning as

The function PUSHJIGQ returns the name of the new jig instance as its value.

Examples:

```
(PUSHJIGQ (GREET 'Person 17) Self Manner Friendly Priority
                                    (Gdistance Self Person 17))
```

(If the returned value is the name "GREET 37", it indicates that the 37th copy of the jig GREET was created.)

```
(PUSHJIGQ (APPLY* (PICK1 '(GREET IGNORE))
                          'Person 17))
```

Jig Sequences

Jig sequences are defined by expressions of the form

"(JIGSEQUENCE <type> <S ... S>)"

where
<type> can take the values POST, PRE, or FREE,
while
<S> is a LISP-expression which (under all conditions) ends up with a call of PUSHJIGQ.

Example:

```
(JIGSEQUENCE FREE
    (PUSHJIGQ (RISE))
    (COND [(IS-MAID)(PUSHJIGQ (MILKING))]
          [(IS-FEMALE)(PUSHJIGQ (PREPARE-
                                 BREAKFAST))]
          [T (PUSHJIGQ(INSPECT-STABLES))])
    (PUSHJIGQ (BREAKFAST)) )
```

JIGSEQUENCE returns the list of the created instances.

The three values of <type> have the following meanings:

FREE indicates that there are to be no succession or precedence links from JNR, the jig instance in which the jig sequence is defined, to any element of the sequence, or vice versa. (That is, 'JNR::Succedes' and

'JNR::Precedes' are not to be modified. That JNR is the 'ancestor' of the sequence elements, is recorded, however, as usual.)

PRE specifies that the defined jig sequence is (in its entirety) to precede JNR; that is: the last sequence element's property 'Precedes' is a list in which JNR's name is an element, and 'JNR::Succedes' contains the name of the last element of the jig sequence.

POST specifies the opposite, namely that JNR is to precede the first element of the jig sequence.

No errors occur if we define sequences with only a single element. JIGSEQUENCE can thus be used to link single instances into a sequence in front or behind JNR. "(JIGSEQUENCE FREE (PUSHJIGQ (....)))" is of course, synonymous with "(PUSHJIGQ (...))".

Interaction

INTERACTION is a 'macro' which, like a PROG in LISP (or like any ALGOL or FORTRAN program), is a sequence of functions, there can be 'labels' which one can 'go' to, and there can be several program 'exists'.

The syntactic form of INTERACTION is

"(INTERACTION <list of local variables> . <body>)"

where
<list of local variables> is a list of variable names which we intend to use within <body>; if the same name occurs elsewhere in the total program, that other occurence has no effect upon the value of the local variable here, which merely happens to have the same name. That means that if we wish to refer to the value which a certain variable has outside of <body>, its name must not be found in the list of local variables;
<body> is a sequence of LISP expressions (to be specified below in greater detail); and the dot "." has again the meaning it had in

Example:

```
(DEFJIG FOO (A B C) (PRIORITY 1 DURATION 1440
                                 XCEVERY 30)
    .
    .
    .
    (INTERACTION (X)
    L0 (SETQ X (F1 A B)) (COND  NULL X) (GO L0) )
    L1 (F2 X) (IA-NEXT L3)
    L2 (F3 X) (IA-NEXT L3)
```

```
L3 (COND [(F4 C) (IA-NEXT L1)]
         [(F5 C) (IA-NEXT L2)]
         [(RANDOMLY) (RETURN)]
         [(RANDOMLY) (IA-NEXT L0)]
         [T (IA-NEXT L3)]))
.
.
.
.
. )
```

L_0, L1, L2, and L3 are 'labels' which either a 'GO-statement' or an 'IA-NEXT-statement' refer to. The important difference between "(GO Lo)" and "(IA-NEXT Lo)" is that the former means a program-execution continuation at Lo during this moment, while the latter also means a continuation of program execution at Lo, but the next time FOO is executed by the atman monitor. "(RETURN)" indicates a definite exit from the interaction body. The values of the local variables are lost after a definite exit.

(IA-NEXT leads to a (non-definite) exit from 'Xql' with the value "Continue".)

We can write "(IA-NEXT Z)", where "Z" is no label but a variable which evaluates to a label.

Feedback

INTERACTION was designed to facilitate the description of interactions between several entities (but it can, of course, also be used for describing the behaviour of a single entity over a period of time).

Interaction is often parametrized by feedback from others. "(REPLY X)", if written in an instance JNR, assigns the value of "X" to the jig property 'Answer' of the 'Ancestor' of JNR. The 'Ancestor' of JNR is in these cases often an instance containing an interaction expression, but feedback can also be of importance in jig sequences. "(ANSWER)", if written within the body of the 'Ancestor' of JNR, evaluates to the reply sent by JNR.

Execution Control

Nogo

The execution of a jig instance JNR must sometimes be inhibited because the test of explicit execution conditions in the jig body shows that execution is to be postponed. The statement "(NOGO)" achieves an exit from the jig body with the value "Postpone"; "Xqstarted' loses its value again and indicates thus that the execution of JNR has not yet started.

Continue

When the expression "(Continue)" is encountered in a jig body during execution, execution is suspended and "Continue" is returned as exit value from 'Xql'. This means that we can prevent an instance from being abandoned even if its 'Duration' has expired.

Xqcontrol

Assume that we wish to stop all other descendants of the "Ancester' of JNR. The expression

```
"(FORALL J IN (DESCENDANTS (ANCESTOR JNR)) DO
     (OR (EQ J JNR)
         (XQCONTROL J 'ABANDON) ))"
```

will achieve exactly this. It will assign the value "Abandon" to the property 'Xqcontrol' of all J except JNR.

The expression "(STOPSEQUENCE J)" means that all jig instances which belong to the sequence to which also J belongs, are to be abandoned (with the help of XQCONTROL). If its successors are to be abandoned, one is to remove 'J::Succedes' first (and also the references to J from its precedents).

Time Points and Time Steps

Before one starts the very simulation process, one can define own time-unit labels; one needs, of course, to indicate the number of basic time units comprised by each of the newly-defined time units. This is done

with the help of the functions

"(timesorts.<L>)" and "(timeprops.<L>)"

where <L> is a list of time-unit labels in the first case, a list of numbers in the second.

Example:

```
(timesorts minute hour day month year)
(timeprops    1    60   24   30    12 )
```

One sees from the example that we have imposed a restriction for ease of specification, namely that each (except the first) time unit's cardinality can be expressed as an integer multiple of the preceding unit's cardinality. (Remark: "Cardinality" means here "number of basic time units in a time unit".)

The example's specifications are tacitly assumed if one omits them in one's program. There are two global variables, 'timesorts' and 'timeprops' (that is, their names coincide with those of the two functions), which from the start have the values "(minute, hour, day, month, year)" and "(1, 60, 24, 30, 12)", respectively. It is these values we change by calling the above-mentioned functions. The current time-step length 'fugit' has the default value 1. The global variables 'minute', 'hour', 'day', 'month', and 'year' are set to zero from the start or when we call the above functions; observe that these variables are not identical in meaning with the time-unit labels which look alike. "hour" as a label identifies a time unit, while "hour" as a variable (that is: when it is evaluated) denotes the current hour of the day. Also 'now' is set to zero from the start or upon a redefinition of the variables 'timesorts' and 'timeprops'.

The function call "(step-up-time)" updates 'now' by adding the value of 'fugit' to it. Simultaneously are (in the default case) 'minute', 'hour', 'day', 'month', and 'year' updated, if necessary. For instance, if the very first instructions in our program were "(fugit 63)" and "(step-up-time)", 'now' would be set to 63, 'minute' to 3, and 'hour' to 1.

In order to facilitate the specification of time intervals, such as "two years, 5 months, and ten days", we have defined a function called 'time' which is used as exemplified by:

"(time 2 'year 5 'month 10 'day)"

'time' expressions evaluate into the corresponding number of basic time units. "(time 1 'hour)" evaluates thus into the number 60; "(time 1 'hour 18 'minute)" into 78; etc. However, by writing "(time 1 'minute)" instead of "1", and "(time 1 'day)" instead of "1440", the user becomes largely independent of the choice of basic time units.

Atman-REC

Atman3, in contrast to Atman2, contains only a single time queue for all agents. All jigs are made 'interaction jigs' (in the terminology of Atman2), that is: jigs are programs that can be suspended and which thus can describe processes that procede for a long time in parallel with other processes.

Torgny Tholerus has also 'stream-lined' the monitoring by advancing time not in minimal steps but directly up to the next scheduled moment of jig activation. Whenever a jig is to be suspended, we indicate for how long, and the jig will be scheduled for next activation immediately after the specified suspension period. Indefinite suspension is possible; we need only specify a condition to be fulfilled by some object. This design is, of course, much more effective than the original one.

Jig Definition

Jig definition is easier than before.

E Example:

```
greet:instance attention (approx 20), priority 8;
    do print ("hello", x, " how do you do?")|
```

The words "instance" and "do" are operator names. Instance fetches and assumes the appropriate jig properties, while do initiates the evaluation of the jig body. The jig body consists of REC expressions.

Jig Sequences

Jig sequences can be made ordinary statement sequences since REC allows 'parallel' functions. However, we indicate jig 'parallelity' by using the operator "&", not the comma, since we are not interested only in the program-control structure created by it.

Parsing rules for '&': "f; g&H; i" is equivalent to "f; (g&h); i"; "f&(g&h)" is equivalent with "f&g&h" and also with "(f&g)&h"; "f; g&(h; i); j" is not equivalent with "f; g&h; i; j".

Example:

```
...; rise; (if is-maid then milking else
                if is-female then prepare breakfast
                else inspect-stables); breakfast;...
```

is a part of a jig body, and implicitly a jig sequence.

All jigs referred to in a jig sequence are in principle assumed to belong to the sequence. In order to detach a jig from the jig sequence we need therefore write "pushjig f". For example, if

```
...; e; pushjig f; g&h; i;...
```

is a part of a jig body, and e, f, g, h and i are jigs, f cannot be pushed before e has been abandoned, but g and h can be pushed as soon as f has been pushed; i, however, cannot be pushed before the 'parallelly' computed jigs g and h have both been abandoned.

A 'competitive' parallel pushing of jigs can be achieved by using the pushing function 'xorpushjig'. (The prefix "xor" alludes to "exclusive or".) This function is used to push several jigs simultaneously, but the jig that is executed first is to suspend the other ones until it is abandoned, whereafter the same rule applies again, etc.

Execution Suspension

Assume that we wish one link in a jig sequence to have a long duration. Atman2 would have checked many times whether the jig's duration had

expired. Atman3 checks not at all; it places instead a signalling task into the time queue which is to activate the next jig in the sequence. Since every jig body is in itself a jig sequence, it is ingeniously simple to suspend jig execution. It is even possible to let the signalling task be conditional.

Suspension is commanded by writing

```
...; wait n;...
```

or equivalently

```
...; suspend jig, n;...
```

where n is an expression that evaluates to a number. "jig" is used in Atman3 to refer to the presently executed jig instance. The second expression is more general; we can namely replace "jig" by any other reference to a jig instance and thus suspend other jig instances.

If we wish to await a certain event m, which is to be caused or experienced by M, we write

```
...; wait for m;...
```

M is given a 'signal table', and an entry in this table that 'jig' is to be activated in case and when M causes or experiences m. m is usually specified in the form of a condition.

Syntax Extensions

Argument Indicators

Atman-REC allows us to define jigs as operators with 'argument indicators'. This is, in a way, a return to formal parameters, even a doubling of them. But since REC is variable-free, it is especially easy to implement this particular feature in it.

Assume a function call "move (a,b,c,d)", where a is an object, b the present location of a, c is the location to where a is to be moved, and d is a condition to be fulfilled. If we have many such functions, it can be difficult to remember arguments orderings. We wish also to omit a and b sometimes, because they are obvious, and d because there

is no condition. In fact, we wish to be able to write "move object a from b to c if d" at one time, "move to c" at another. The orderings should also be variable; "move from a to b" should be synonymous with "move to b from a".

To allow such flexible notation, we need merely to define an operator '::' which automatically defines how the whole expression is to be transformed into its canonical form. The function will be associated with a small table which specifies the envisaged indicators (such as 'to' or 'from'); these indicators can, within the function body, be used for making references to the values they indicate.

That means that we merely need to write

```
move (from, to, object, if, by):: .... |
```

instead of

```
move: .... |
```

in order to obtain flexibility. The efficiency cost is negligible.

The Atman3 Database

It is important that the pointing to a particular item in the database is easy and of obvious and unambigious meaning for the program reader also, not only for the computer. We wish therefore to be able to write "the door of the house beside the house of John is green" instead of

```
"a:=tabx(John, 'house); b:=tabx(a, 'beside);
     c:=find(a, 'house);
     if c then tabx (c, 'door):='green".
```

We wish also to be able to ask for information similarly: "what is the color of the door of the house beside the house of John" and "is it true that the door of the house beside the house of John is green". The first question requires that we have said earlier in some connection that 'green' is a color; for example, when we said "the car of the person named John has the color named green".

Is it a particular fact or a general rule that the door of the house beside the house of John is green? If we do not decide the question explicitly, Atman3 will assume that it is merely a fact, not a rule. But we can write "rule:the door ..." or "fact: the door..." in order to be explicit on this matter.

Our experience with the programming of jigs for Atman2 tells us that a major portion of all such programming consists of searching certain information items, or of pointing to and updating them. Programming will surely be more fun as well as more efficient if these things can be more conveniently specified.

The updating of items is uniquely easy. For example, let us increase the family of John by one person; we write "(x:=x+1) the size of the family of John". Note that "(x:=x+1)" is to be understood as a function which has "the size of the family of John" as its argument; this input evaluates the input list to (x:=x+1), while 'x' is the function that picks the first element from its input list. It is obvious that arbitrarily complicated updating rules (conditional ones, for example) are equally easy to formulate if we deal with a direct or an indirect item reference.

The difference between rule and fact is more obvious when we have propositions such as "every door of every house beside every house of John is green". Assume that John owns two houses and buys a third. If the above proposition expressed a fact, nothing follows from the house purchase, but if it expressed a rule, we are to change the colors of all doors of all houses beside the new house of John to green if they are not green already.

Propositions that contain "a" are often more difficult to interpret. Take "fact:every door of a house beside the house of John is green", for instance. If no stored information iten can be found that represents a house of John, such an item is to be created. If there are houses owned by John, none fulfilling the stated condition, we have two possibilities at hand: if all houses are explicitly said to have doors that are red, blue, etc, but not green, we are to create a new house which fulfills

the condition. But if we cannot ascertain that the condition is not fulfilled, we must register a category that comprises all houses beside the house of John; this category is associated with the condition. Each time any member of that category changes or is given a door color, we must check whether we now are able to draw more definite conclusions that render the category registering superfluous. (Note that we thus again avoid the frame problem.)

If we do not wish to have a new house created but instead want to have one of the houses beside John's house change its colors, we must write "change: every door of a house beside the house of John is green". A random selection is made. If we want to specify more exactly which house we mean, we need to point to it directly or to a subset of John's neighborhood.

Such pointing is done with the help of the function 'find ...such that': "find every house beside the house of John such that it is white, is old, has owner named Jim or has owner named Joe".

We wish also to be able to store facts or rules about relationships. For example: "fact:the house of John is more green than the car of Jeannine". This implies the storage of the facts that John has a house, that it is green, that Jeannie has a car, that it is green, and that a comparation relation exists between the car's color and the house's. More general facts will also be stored (if they are not already, of course), namely that persons can have houses and cars, and houses and cars can have colors.

"rule: a house of a person is more large than a house of a dog" implies similarly that dogs can have houses. Adjectives are treated as names, belonging to a sort: 'green' was specified to be the name of a 'color'; 'large' must simultaneously be specified to be the name of a 'size'. (This can be done by writing "(large has sort named size; large)" instead of "large" in the rule above.) Such information will be valuable when Atman3 is to try to 'solve problems'.

The above rule does not refer to concrete houses, persons, or dogs, but

to categories. The comparation relation holds between the two categories 'a house of a person' and 'a house of a dog'.

Hypothetical Reasoning

Bobrow and Wegbreit (ref 4) have proposed an implementation design for the control of multiple data-base versions that allows a programmer to work effectively with several data-base versions simultaneously. Torgny Tholerus has already augmented REC with this new control structure, and we can now expect to be able to let simulated agents perform hypothetical reasoning.

Let us exemplify the power of the new control structure with a description of how the eight-queens problem (cf ref 4, p 595) can be solved: The task is to place "eight queens on a chessboard in such a way that no two can take each other". The function 'conflict' which we have omitted here, checks whether a queen in square s can take any of the already positioned ones.

```
n:=0; s:= nil;
loop: k:= select (1--8)
      check conflict (s,1);
      n:=:n+1; s:=:par(1,s);
      if n=8 then s else loop |
```

The operator ":=:" indicates (and sees to it) that the assignment is reversible in case backtracking becomes necessary.

(Note that the eight-queen problem can be regarded as a simple paradigm for role-taking in a social context.)

References

1, Holstein. HOMO CYBERNETICUS, Vol 1-5, Sociografica, Uppsala.

2. Risch. GETT/PUTT Programdokumentation, Datalogilaboratoriet, Uppsala.

3. Risch. MAKROEXPAND, DLU 73/22, Datalogilaboratoriet, Uppsala.

4. Bobrow & Wegbreit. A model and stack implementation of multiple environments, CACM, vol 16, nr 10, October 1973.

15. 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 inpulser från den nya generation av "ultraspråk", som QLISP (se kapitel 4d).

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 dels ett system som simulerar patience-läggning, där patiencerna beskrivits i naturligt språk, dels ett importerat system, som besvarar frågor ställda i naturligt språk.

15a PALLE - ett system som lägger patience

All slags programmering kan man betrakta som beskrivning av uppgifter som man förelägger en dator. Datorn svarar med resultat eller felutskrifter. Vid ordinär programmering är det sätt man beskriver uppgifterna på strikt reglerat genom det programmeringsspråk man använder och dessutom måste man explicit tala om hur uppgiften skall lösas, och denna beskrivning av uppgiften måste specificera algoritmen in i minsta detalj.

I moderna högnivåspråk av typ QLISP (jfr 4d) behöver inte användaren bekymra sig om viss detaljplanering av algoritmen, men han är fortfarande tvungen att arbeta innanför språkets syntaktiska råmärken.

Vad man naturligtvis eftersträvar är att få formulera uppgiften på det sätt som faller sig naturligast, utan inskränkningar på grund av att det

är en dator som skall förstå beskrivningen. Man vill med andra ord kunna kommunicera med datorn i naturligt språk.

Inom naturligt-språk området har vi nu startat ett projekt som syftar till att bygga ett system som tar emot beskrivningar av uppgifter i naturligt språk och sedan kan utför uppgifterna på begäran. Som problemområde används patience-läggning, vilket verkar vara ett område med flera intressanta aspekter samtidigt som begreppsvärlden tycks överkomligt stor.

När man betraktar en beskrivning av hur man lägger en patience kan man finna vissa karakteristiska drag. Texten består huvudsakligen av uppmaningar att göra vissa saker i tur och ordning eventuellt förbundna med olika villkor. Bland dessa uppmaningar finns dessutom ibland konstateranden att nu gäller det och det. Dessa konstaterande är en kontroll att man förstått beskrivningen så långt och utfört vas som skall utföras. Vidare kan det finnas exempel som talar om vad man skall göra i just detta fall. Ett sådant exempel kan då också användas för att kontrollera att beskrivningen är rätt uppfattad. Dessa tre typer av meningar innehåller alla information som är relevant för utförandet av uppgiften. Dessutom kan det finnas annan information som är helt oväsentlig när man skall lägga patiencen, t ex kan det finnas en historik som berättar vem som hittade på patiencen och dylikt. Sådan information är vi för närvarande inte intresserade av.

Systemet består i huvudsak av tre moduler:

Nätverksparsern med tillhörande grammatik.
En semantisk analysator.
Patience-läggaren.

Nätverksparsern

Nätverksparsern har beskrivits i tidigare årsrapporter, varför vi hoppar över den här. Den grammatik som används är en modifierad variant av den som använts tillsammans med SEPAC-paketet (jfr DLU -74).

Vid analysen i nätverksparsern omstruktureras meningen och man får en

syntaktisk kasus-lista.

Ex: Vänd upp tre kort i taget ur talongen.

VERB VÄNDA-UPP
OBJ (3 KORT)
I TAG
UR TALONG

Strukturen med syntaktisk kasus-lista, SYNKAL, är en förgrovning av den representation som SEPAC använder och medför att viss information kan gå förlorad. Jag bedömer dock inte detta vara någon större fara, utan anser att nackdelarna uppvägs av SYNKALS:s större enkelhet.

Den semantiska analysatorn

Den semantiska analysatorn SEEMAN, mottar en syntaktisk kasus-lista som inmatning och genererar av denna en semantisk kasus-lista.

Ex:
VERB VÄNDA-UPP
OBJ(3 KORT)
I TAG → (VÄNDA-UPP I-TAGET 3)
UR TALONG

SEEMAN utgår från verbet i den syntaktiska kasus-listan. Varje verb har förbundet med sig en funktion som går igenom SYNKAL-formen element för element och bygger upp en SEMKAL-form. I SEMKAL finns det en kasus-struktur för varje verb, dvs i SEMKAL har ett visst kasus vid ett visst verb en bestämd semantisk betydelse. Alla kasus behöver dock inte finnas med.

Patience-läggaren

Patience-läggaren, PALLE, är den del av systemet som utför de uppgifter som begärts, dvs simulerar att den lägger patiencen. PALLE har ett bord till sitt förfogande som är uppdelat i rader och kolonner där han kan lägga ut kort. Vidare har han en hand dit han kan ta upp kort. De kort han inte lagt ut kallas talongen.

Palle interpreterar uppgiften som den ser ut i SEMKAL och utför de kommandon som ges.

I och med att PALLE är en interpretator måste han förstå sina kommandon. Han vet alltså vad en sekvens är, vilka kort som har samma färg, att en knekt är högre än en åtta etc.

Exempel

Nedan följer exemple på en patiencebeskrivning dels i naturligtspråk, dels i SYNKAL- och idealiserad SEMKAL-form. Exemplen lämnas okommenterade eftersom en kommentar skulle föra alltför långt in på en teknisk beskrivning.

Naturligt språk:

Lägg ut sju högar med fem kort i varje.
Vänd upp tre kort i taget ur talongen.
Man får bygga på korten på bordet i fallande alternerande sekvens.
Om det kommer fram ett ess lägger man upp det.
Sedan bygger man på essen i sekvens i samma färg.

SYNKAL-form:

```
VERB LÄGGA-UT
OBJ (7 HÖG)
MED (5 KORT)
I VARJE

VERB VÄNDA-UPP
OBJ (3 KORT)
I TAG
UR TALONG

VERB FÅ
BY MAN
OBJ VERB BYGGA
    PÅ KORT
    PÅ BORD
    I (ANDPROP FALLANDE ALTERNERANDE SEKVENS)
```

```
IF VERB KOMMA-FRAM
    BY ESS
THEN VERB LÄGGA-UPP
    OBJ DET
    BY MAN

VERB BYGGA
MODE SEDAN
BY MAN
PÅ ESS
I SEKVENS
I (SAMMA FÄRG)
```

SEMKAL-form:

```
A1  (LÄGGA-UT HÖG 7 KORT 5)
A2  (VÄNDA-UPP I-TAGET 3)
A3  (BYGGA REL (ALTERNERANDE FALLANDE-SEKVENS) LOC BORD)
A4  (IF (KOMMA-FRAM REL ESS)
    THEN (LÄGGA-UPP OBJ (AKTKORT)))
A5  (BYGGA LOC ESS REL (SEKVENS SAMMA-FÄRG))
```

Planer för 1975

Systemet finns implementerat ungefär så långt som beskrivits i tidigare avsnitt. I det fortsatta arbetet kommer vi att lägga in interaktivitet så att systemet kan delta mera aktivt och fråga om oklarheter och dylikt. Detta förutsätter dock tillgång till ett interaktivt INTERLISP-system. Andra problem är analysen av sambandet mellan olika meningar i beskrivningen, lösande av tvetydigheter och metoder att försöka hitta en ny analys i något steg om någon inkonsistens dyker upp. Vidare återstår mycket utveckling av det grundsystem som beskrivits här.

15b. Studium av SCHOLAR

För att närmare studera problemlösningar i ett större frågebesvarande system för naturligt språk har det vid BBN utvecklade. programmet SCHOLAR överförts till DLU. Det har då samtidigt kommit att (ofrivilligt)

fungera i som ett testprogram för INTERLISP/360-370, eftersom det fungerar i INTERLISP/PDP och utnyttjar en mycket stor del av INTERLISP:s faciliteter (varav vissa har fått kortslutas, eftersom de ännu inte implementerats i INTERLISP/360-370).

SCHOLAR är ett frågebesvarande och frågeställande system skrivet av J.R. Carbonell (1). Databasen i den aktuella versionen är Syd-Amerikas geografi, men programmet uppges även fungera för andra, annorlunda databaser.

Den frågeställande delen, som inte kommer att behandlas närmare här, bygger på en dynamisk frågegenerering, närmast påminnande om en muntlig tentamen, och alltså inte av typ programmerad undervisning, vilket annars är normalt vid frågeställande datorsystem.

Inmatningsbehandlingen

Den del av systemet, som sysslar med behandling av inmatningen i naturligt språk, har ett speciellt intresse genom att den fungerar utan användande av någon syntaktisk parser av typ Woods nätverksparser (se 11a). Den är vidare starkt influerad av att inmatningen antingen är en fråga eller ett svar.

Redan i inläsningrutinerna skiljer SCHOLAR av skiljetecken och slår ihop flerordsuttryck till vardera ett "ord". Flerordsuttrycken kan här vara av typen RIO DE JANEIRO men även av typen WHAT IS. Speciellt igenkännes de fraser som normalt inleder en fråga (även fraser av typ "TELL ME ..." betraktas i SCHOLAR som frågor). Sammanslagning sker endast om ingen risk för felaktig sammanslagning finnes, men, å andra sidan, även av så långa uttryck som BE CONSIDERED A PART OF.

Det första analysrutinerna gör är att avgöra om det rör sig om en fråga eller ett svar. Väntar sig SCHOLAR en fråga, utgår det från att allt, som kan tolkas som en fråga, är det, väntar det sig ett svar, testar det ändå om det kan vara en fråga, men tar hänsyn till vaga, frågeliknande svar av typ "MAYBE LARGE?". SCHOLAR börjar nu leta efter ord som kan

avgöra frågans typ. Om ett komparerat adjektiv ingår, utgår SCHOLAR från att det har med en jämförelse att göra, och söker därför reda på de två första substantiven, egennamnen eller motsvarande (x och y). Om dessa pekar på element av samma typ (städer, länder etc), jämförs värdena på tillämplig egenskap, i annat fall söker SCHOLAR reda på vilket x av y, som har största värdet (eller lägsta om adjektivet är sådant) på den egenskap adjektivet åsyftar. På motsvarande sätt testar sedan SCHOLAR på andra tecken till att en jämförelse önskas. Om så inte är fallet söker SCHOLAR nu efter konjunktioner och delar i så fall upp frågan på flera likstrukturerade frågor och börjar om från början. Här är SCHOLAR dessvärre ganska lätt att "lura". Först härefter undersöker SCHOLAR om det verkligen känner till alla ord (i de tidigare konstruktionerna har detta inte varit av intresse), och om så är fallet vidtager klassificeringen av frågan. Denna sker, liksom ovan, genom att SCHOLAR söker efter vissa intressanta ord av typ DO, HOW BIG, WHAT etc.

Den klassificerade och uppspjälkade frågan överlämnas nu till SCHOLAR:s informationssökningsdel. Denna försöker först hämta informationen med den för frågeklassen naturligaste permutationen av objekt, egenskap och värde. Misslyckas detta, försöker SCHOLAR med andra tänkbara permutationer, eventuellt kombinerade med en mer sofistikerad slutsatsdragning (än bara hämtning). Om SCHOLAR i något av dessa försök hittar ett svar, utgår det från att det tolkat frågan på rätt sätt och skickar detta vidare till språkgeneratorn. I motsatt fall ger SCHOLAR upp.

Ifall inmatningen utgör ett svar, gör SCHOLAR en relativt enkel matchning mellan det inmatade svaret och det svar SCHOLAR själv genererat samtidigt med frågan.

Informationslagringen

All inläggning av information måste ske "för hand"; systemet har inga faciliteter för att ta emot information vare sig i naturligt språk eller via speciella funktioner.

Exempel på lagrad information (under elementet ATACAMA\DESERT)

```
(((XN ATACAMA\DESERT ATACAMA)
  (DET THE \)
 NIL
 (SUPERC NIL DESERT REGION)
 (SUPERP (I 6 B)
         SOUTH\AMERICA
         ($EX CHILE PERU))
 (LOCATION NIL (IN NIL ($EX CHILE PERU))
           (BETWEEN NIL ($EX PACIFIC ANDES\MOUNTAINS)))
 (CLIMATE (I 3)
         TEMPERATE
         (DRY NIL EXTREMELY)
         (TEMPERATURE (I 6 B)
                      (RANGE NIL 40 90))
         (PRECIPITATION (I 6 B)
                        (RANGE NIL 0 4)))
(TOPOGRAPHY (I 2)
            (SOIL NIL BARREN ROCKY SANDY))
(PRODUCTS (I 1)
          (MINERALS NIL IMPORTANT
                    (PRINCIPAL NIL
                               ($L COPPER IRON NITRATES SILVER
                                                    SULFUR
                                                   ZINC)))))
```

Lagringen är som synes ganska lättläst (åtminstone om man är van vid LISP) och även ganska lättskriven. Detta är avhängigt av att inga restriktioner finns på vilka ord som får finnas som attribut och värden, även om sådana ord som systemet har t ex slutsatsdragningsregler för, givetvis säger mer än andra. I början av varje element ligger den ordklassinformation som trots allt behövs, framför allt för utmatningen, (här att ATACAMA\DESERT är ett substantiv av egennamnstyp) och olika stavningar på elementet (\ skrivs ut som blank). Detta följs direkt av utskriftsinformation för språkgeneratorn. Resten är egenskap-värde-"par"

till objektet. Varje egenskap kan ha flera värden, och dessa i sin tur kan vara nya egenskap-värde-par. Denna rekursivitet är en av orsakerna till att lagringen inte skett på vanligt LISP-sätt med PUT, GETP etc.

Platsen omedelbart efter egenskapen är reserverad för olika flaggor som SCHOLAR använder. Den vanligaste av dessa är I-flaggan (för Irrelevancy). Den berättar om hur pass ointressant upplysningen är, och används dels för att undvika alltför meningslösa alternativ i frågegenereringen, dels för att få fram den intressantaste upplysningen vid frågor av essätyp. En annan flagga (B) utvisar information som främst är intern och som normalt inte skall skrivas ut.

Informationssökningen

Det man önskar göra vid informationshämtningen är att komplettera objekt-attribut-värde-tripplar där noll, ett eller två av trippelns element är okända. Följande frågetyper är vettiga:

(i)	(objekt egenskap ?)	Could you give me the location of Iquique?
(ii)	(objekt ? värde)	What is the connection between Peru and Atacama?
(iii)	(? egenskap värde)	Of what country is Brasilia the capital?
(iv)	(objekt ? ?)	Tell me something about the Mato Grosso.
(v)	(objekt egenskap värde)	Is the Parana a tributary of the Rio de la Plata?

Frågor av typ (i) and (ii) besvaras på likartade sätt, nämligen genom att (med hjälp av INTERLISP:s editor) i objektet söka reda på den givna egenskapen resp värdet. Ur den nästning av LISP-uttryck. som detta förekommer i, bildas sedan svaret. Typ (iii) är svårast att besvara. I ovanstående exempel får SCHOLAR söka efter (CAPITAL ... BRAZIL ....) på alla COUNTRYn det känner till. Än värre är frågan "Brazilia is the capital of ---.", där SCHOLAR måste undersöka allt som kan ha en CAPITAL (dvs COUNTRY och STATE). I fall (iv) lämnas all information som finns lagrad under objektet. Hur mycket av detta som skall användas i svaret bestäms av frågans typ (TELL ME SOMETHING, EVERYTHING, MORE etc ABOUT ...)

i samverkan med ovan nämnda I-flaggorna. Om inte EVERYTHING eller likn. finns med, besvaras frågan till ett visst "I-djup" som sedan komms ihåg, så att en fråga om ytterligare information för samma objekt besvaras med ett nytt "I-intervall". Frågor av typ (v) slutligen, besvaras som typ (i), med tillägget att det givna värdet jämföres med det erhållna.

Naturligt-språk-generatorn

Naturligt-språk-generatorn i SCHOLAR är byggd för att på ett så enkelt sätt som möjligt få en begriplig och naturlig utmatning. Ett resultat av denna enkelhetssträvan är att BE är i stort sett det enda verb som användes. Detta återspeglar databasen, där såväl attribut som värden är substantiv, adjektiv och motsvarande. Vidare används i stor utsträckning konstruktioner med OF (the A of B is C). För att undvika upprepningar håller generatorn reda på vad den just håller på att mata ut och ersätter, där så är möjligt, substantiv och egennamn med pronomen. Då utmatningen genereras är inmatningen för länge sedan glömd av SCHOLAR och det finns ingen som helst koppling av den typ, som förekommer t ex i ELIZA.

Några exempel på in- och utmatning:

```
*Tell me about Cordoba.*

     CORDOBA IS A CITY.
       IT IS IN NORTHERN ARGENTINA.

*Which is the biggest city in Brazil?*
  SAO PAULO IS THE BIGGEST CITY.

*Tell me more about Cordoba.*

  THE POPULATION OF CORDOBA IS APPROX 700,000 PEOPLE.

*Compare Dutch Guiana and French Guyana.*
     THE SIMILARITIES ARE:

  THEY ARE BOTH COUNTRIES.

    THEY ARE IN NORTHERN SOUTH AMERICA.

  THE PRIMARY DIFFERENCE IS:

  THE AREA OF SURINAM IS LARGER BY A FACTOR OF 1.5.
```

Det finns givetvis fortfarande åtskilliga konstruktioner som SCHOLAR inte klarar av, men i de flesta fall verkar det förhållandevis lätt att lägga in kod som klarar dessa utan att andra delar av systemet påverkas. En del av denna kod har också skrivits och införts i SCHOLAR vid DLU. Andra konstruktioner, t ex konjunktioner, är svårare att komma till rätta med, bl a på grund av bieffekter, men verkar inte oöverkomliga. På det hela taget ger SCHOLAR intrycket att ett större system med naturligt-språk-inmatning kan fungera bra utan syntaktisk parser. Det skall dock observeras att SCHOLAR är begränsad till i stort sett en satstyp, nämligen frågor, och även där ganska enkla satser.

Referenser

1. Jaime R. Carbonell: Mixed-Initiative Man-Computer Instructional Dialogues, BBN Report No 1971, 1970.

16. ÖVRIGT

16a. NAMN, program för litteraturreferenser

NAMN är ett program, som utförts som trebetygsarbete vid institutionen för informationsbehandling. På en direktfil lagras för varje rapport, författare, rapportnamn, årtal samt kommentarer om rapporten. Programmet kan generera litteraturreferenslistor, sådana som förekommer i slutet på en rapport. Samtidigt skall man kunna interaktivt fråga om en rapport eller om ett antal rapporter en viss författare skrivit. Kommandon finns för att uppdatera och utöka filen.

16b. Program för utvärdering av hashmetoder

Detta program, kallat HASHTEST (DLU 74/33) skrevs och dokumenterades av Kerstin Cohen och utgjorde examens arbete för 3 betyg i informationsbehandling.

Programmet testar effektiviteten hos

- hashfunktioner
- hashfunktioner i kombination med någon tabellorganisation.

Input till programmet är

- Den (de) hashfunktioner som skall testas (i form av FORTRAN anropbara subrutiner)
- De symboler man önskar ha som test material
- Styrkort för val av tabell organisation, statistikutskrifter, olika hashsituationer etc.

Programmet organiserar sedan körningen, hanterar hashtabeller och genererar och rapporterar statistik av körningen.

B NYA PROJEKT

17. UTVECKLING AV DISKRIMINERINGS-DATABASER

Bakgrund.

Under våren 1974 har vid Datalogilaboratoriet en version av QLISP-språket (1) implementerats (2). QLISP, liksom andra AI-språk tillhandahåller en databas där användaren ges möjlighet att lagra väsentligen godtyckliga uttryck samt en mängd verktyg för att manipulera uttrycken och för att göra slutsatsdragning i databasen. QLISP-databasen möjliggör dessutom lagring och manipulering av mängder, både ordnade (tuple, vector) och oordnade (bag, class). Denna databas är organiserad som ett diskrimineringsträd och denna representation passar väl för den typ av sökning man oftast vill göra i QLISP, nämligen associativ sökning. Denna organisation har emellertid visat sig vara otillfredsställande med avseende på effektivitet och flexibilitet. Bl a kan man från beskrivningen i (2) utläsa att trädets utseende är beroende av ordningen i vilken uttrycken har lagrats. Dessutom är sökalgoritmen såväl som diskrimineringsalgoritmen bestämda en gång för alla, vilket lätt ger upphov till ineffektiva "obalanserade" trädstrukturer.

Projekt

Vid Datalogilaboratoriet har vi i samband med implementering av QLISP studerat olika problem förknippade med diskrimineringsdatabaser och tittat på lösningar och metoder föreslagna i litteraturen (3, ...). Vi ämnar att, med början i höst och förhoppningsvis inom ramen för ett långtidsprojekt, arbeta med utveckling och studier av metoder för att konstruera kraftfulla smådatabaser. Arbetet föreslås i första hand innefatta följande aspekter.

a. Diskriminerings-funktioner

Tillsammans med varje nod i databasen associeras (i många fall implicit) en diskriminering funktion. Vid sökningen (och lagring) kommer sökalgoritmen att kunna hämta information om hur den skall förfara, inte bara från det argumentet som initierade sökningen utan dessutom från varje steg i sökprocessen. Varje nod kommer lämpligen att innehålla mer och mer specialiserad anvisning om hur den skall genomlöpas. Diskriminerings-funktioner

skall antingen kunna genereras automatiskt av systemet eller ges av användaren. Diskriminerings-funktioner kombinerar således på ett naturligt sätt data och program till ett kraftfullt system för att dels beskriva databas-strukturen och dels för att utföra bearbetningar i databasen.

b. Balansering, (dynamisk) omorganisering

Effektiviteten hos en diskriminerings-databas är starkt beroende av dess tilltänkta användning. Även topologiskt optimalt balanserade träd (5) kan vara mycket ogynnsamma för praktiska tillämpningar. Forskning pågår på olika håll med självorganiserande databaser, där ett av problemen är att finna algoritmer för själva omorganisationen. För detta behöver man t ex kunna samla information (ex statisk information) under en körning. Denna information kan antingen användas under körningen för dynamisk omorganisering, eller ligga till grund för en senare omorganisering av databasen. I vårt system vill vi lägga upp en mekanism i diskrimineringsnätet för att samla information som i första hand kommer att användas för att utvärdera sök- och lagrings-algoritmer. Denna mekanism hoppas vi också skall bli ett bra verktyg för att upptäcka bättre algoritmer för omorganiseringar. Sådana algoritmer behöver inte innebära en fysisk omorganisering av noderna i databasen utan det kan räcka med att ändra eller generera nya diskriminerings-funktioner.

c. Implicit lagring

Genom att vi tillåter att beskrivande information i procedurform finns med i noden, behöver ofta vissa rekords aldrig lagras (eller sökas efter) explicit, om man kan beskriva deras generella utseende och egenskaper t ex i form av ett mönster eller med hjälp av en funktion. En intressant sidoeffekt av diskriminerings-databasen med diskriminerings-funktioner är att denna kan användas som ett slags filter. Man vill nämligen ofta i slutnoderna lagra bara en hänvisning till var rekordet finns (skall lagras). På så sätt blir diskriminerings-trädet en "liten" men kraftfull databas med möjlighet att utföra slutsatsdragning, men där man i första hand är intresserad av en "approximation" av var en rekord är lagrad (pekare i minnet, post på disk etc).

d. Kommunikation med användaren

Vi kommer att behöva en del programredskap huvudsakligen för att sköta interaktionen mellan systemet och användaren. Beskrivning av databaser är

en mycket viktig del av projektet, vi vet redan nu att datatyper såsom egenskapslistor, ordnade och oordnade mängder, kommer att vara tillgängliga i det planerade systemet (utöver de standard-datatyper som allmänt finns i programmeringsspråken). Vi vill dessutom utreda vilka andra datatyper som är av intresse för eventuella tillämpningar samt ge möjlighet att definiera egna datatyper för att beskriva databasen.

Sammanfattning

Vi ser inte diskriminerings-databaser av den typen vi föreslår som ett lagrings-system för mycket stora datamängder utan snarare som ett mellanled mellan olika tillämpnings-system och den stora datamängden. Vi är övertygade om att smådatabas-forskningen inom en snar framtid kommer att vara till nytta för den traditionella databashanteringen. Vi hoppas också vinna erfarenhet i sök- och lagrings-strategier med tyngdpunkt på associativ sökning, samt i metoder för att representera kunskap.

Referenser

(1) Reboh, René & Sacerdoti, Earl. A Preliminary QLISP manual. SRI AI Center Technical note 81, August 1973.

(2) Datalogilaboratoriet 1975 (årsrapport), sektion 4d (ännu ej utkommen)

(3) Haraldson, Anders. PCDB - A Procedure Generator for a Predicate Calculus Database, Information Processing 74, IFIP, Stockholm 1974.

(4) Knott, D. Gary. A Balanced Tree Storage and Retrieval Algorithm. Proc.Symp. on Information Storage and Retrieval. Univ. Maryland, College Park, Md, 1971, p 175-96.

(5) Knuth, E. Donald. The Art of Computer Programming, Vol. 3, Sorting and Searching. (Särskilt 6.2.3, 6.3)

(6) Rivest, L. Ronald. Analysis of Associative Retrieval Algorithms. IRIA Laboria, Rapport de research nr 54, February 1974.

(7) Rivest, L. Ronald. On Self-organizing Sequential Search Heuristics. IRIA Laboria, Rapport de research nr 61, March 1974.

(8) Sacerdoti, Earl. Planning in a Hierarchy of Abstraction Spaces. SRI AI Center. In Advance papers of the Third International Joint Conference on Artificial Intelligence, August 1973.

(9) Severance, G. Dennis. Identifier Search Mechanisms: A Survey and Generalized Model. Computing Surveys, vol. 6, no. 3, September 1974.

B I L A G A 1

ARBETSFÖRDELNING OCH SPONSORER

Här följer en uppställning som motsvarar innehållsförteckningen, vari angivits vilka som arbetat med uppgiften, samt anslagsgivare eller motsvarande. Uppställningen avser endast 1974, ej tidigare år och ej planer för 1975. Använda förkortningar förklaras sist.

4	Programmeringssystem för smådatabaser			
	4a	INTERLISP/360-370	Jaak Urmi	UDAC
	4b	PLAST	Olle Willén	FOA Hel
	4c	REC	Torgny Tholerus	RJF, STU
	4d	QLISP	René Reboh	IBM, ITM
			Pär Emanuelsson	STU
			Anders Haraldson	STU
	4e	SLISP - Simulering med LISP som programmeringsspråk	Mats Nordström	FOA Hel
5	Tillförlitlighet i programmeringssystem		Anders Beckman	STU
			Jerker Wilander	STU
6	Teori för programmeringsspråk			
	6a	SIMLISP	Mats Nordström	FOA Hel
7	Manipulation av FORTRAN-program			
	7a	SUGFOR - syntaktisk socker för FORTRAN		
			Anders Beckman	STU
			Tom Smedsaas	-
	7b	COMLIST	Per Haeggström	3-betyg
9	Manipulation av LISP-program			
	9a	REDFUN	Östen Oskarsson	STU
	9b	REDCOMPILE	Lennart Beckman	NFR, STU
	9c	INTERLISP simulator	Pär Emanuelsson	Sommarskolan
	9d	TRANS, ett paket för översättning från INTERLISP till LISP 1.5	Pär Emanuelsson	-
10	Programredskap för informationsstrukturer på lägre nivå			
	10a	PCDB - Predicate Calculus Data Base	Anders Haraldson	STU
	10b	DABA - Ideas about management of LISP data base	Erik Sandewall	UU
	10c	TOP	Torgny Tholerus	FOA Hel

11 Programredskap och metoder för naturligt-språknivå

11a	Nätverksparsern	Mats Cedvall	NFR

13 Tillämpningar och experiment

13a	Frågesystem för en ekologisk databas	Sture Hägglund	IBM
13b	VARKAT-DLU. Ett LISP program för simulering av ett interaktivt databassystem vid SCB	Östen Oskarsson Sture Hägglund	STU STU
13c	IDECS. Ett program-paket för pilot-programmering och utveckling av interaktiva informations-system	Sture Hägglund Östen Oskarsson	STU STU
13d	Tillämpnings-experiment med användning av SLISP	Mats Nordström	FOA, Hel
13e	LME kopplingsschema	Mats Nordström	-
13f	Kontroll av tågtider	Mats Nordström	-
13g	BNF - Generering av testdata	Mats Nordström	-
13h	LISP-IMS MANAGER, ett system för strukturbeskrivning av IMS-databaser	Tore Risch	UDAC

14 Användningar i annan forskning

14a	Simuleringssystem för ATMAN	Tore Risch	RJF
		Torgny Tholerus	RJF
		Hans-Jürgen Holstein	RJF

15 Naturligt-språk-arbete

15a	PALLE - ett system som lägger patience	Mats Cedvall	NFR
15b	Studium av SCHOLAR	Klas Mårtensson	FOA

16 Övrigt

16a	NAMN, program för litteraturreferenser	Stefan Holmgren	3-betyg
16b	Program för utvärdering av hashmetoder	Kerstin Cohen	3-betyg

B. Nya projekt

17	Utveckling av diskriminerings-databaser.	René Reboh	

ANVÄNDA FÖRKORTNINGAR

FOA Hel	Försvarets forskningsanstalt, Huvudenhet 1
IBM	IBM:s FUSAM-kommitté (Forskning-Undervisning-Samverkan)
ITM	Institutet för tillämpad matematik
NFR	Naturvetenskapliga forskningsrådet
RJF	Riksbankens Jubileumsfond
STU	Styrelsen för teknisk utveckling
UDAC	Uppsala datacentral
UU	Uppsala universitet
Sommarskolan	Sekretariatet för nordiskt kulturellt samarbete

När anslagsgivare markerats med - har arbetet gjorts utan viss anslagsgivare och med marginalresurser.

FÖRTECKNING över tekniska rapporter 1974

DLU 74/1 Tore Risch: PMG - en Program-Manipulator-Generator i LISP, jan 1974.

DLU 74/2 Johan Sandwall: PREPROC, 1974-02-04 (trebetygsuppsats)

DLU 74/3 Östen Oskarsson: Introduktion och hjälpreda till GIP/GUP, 1974-02-11.

DLU 74/4 Bo Elofsson: FORTIM: ett program för tidsanalys av Fortran-program, 1974-02-15. (trebetygsuppsats)

DLU 74/5 Östen Oskarsson: GIP1, GUP1 - Förenklad in- och utmatning av egenskapslistor, 1974-02-22.

DLU 74/6 Lars Lidén: För-hashning, 1974-02-21. (trebetygsuppsats)

DLU 74/7 Erik Sandewall: Datalogilaboratoriets programbibliotek för smådatabaser (översiktspärm), 1974-02-28.

DLU 74/8 Mats Nordström & Torgny Tholerus: A Parsing Technique Applied to the Programming Language Reduce, 1974-02-28.

DLU 74/9 Lennart Beckman: REDCOMPILE, 1974-04-01.

DLU 74/10 Tore Risch: LISPHELP - ett hjälppaket för LISP 1.6, 1974-04-29.

DLU 74/11 Tore Risch: DEC-MAKRO - ett programmanipulerande program för LISP 1.6, 1974-04-29.

DLU 74/12 Erik Sandewall: Programdok: SAVE - Administration av paket av LISP-funktioner, 1974-05-09.

DLU 74/13 Erik Sandewall: Programdok: FILOP - Diverse operationer på LISP-filer, 1974-05-08.

DLU 74/14 Mats Cedvall: Dokumentation av en kompilator och tillhörande runtime-system för nätverksgrammatiken, 1974-05-28.

DLU 74/15 Anders Haraldson: LISP-details, 1974-06-01.

DLU 74/16 Olle Willén: Mini manual för PLAST users, 1974-06-01.

DLU 74/17 Anders Haraldson: Short PCDB Guide, 1974-06-01.

DLU 74/18 Lennart Beckman: Automatisk optimering av LISP-program, maj 1974, UPTEC 74 33 E.

DLU 74/19 Lennart Beckman: TABLEOUT, maj 1974.

DLU 74/20 Erik Sandewall: Program structure and process look-ahead in language understanding programs, 1974-07-05.

DLU 74/21 Bjarne Däcker: Documentation of the TTT program, Program-dokumentation, 1974-06-14.

DLU 74/22 Erik Sandewall: Documentation of the HIERPROP program, Programdokumentation, 1974-06-12.

DLU 74/23 Stefan Holmgren: NAMN: ett program för litteraturreferenser, 1974-07-15, (trebetygsuppsats).

DLU 74/24 Mats Nordström: Förslag till mall för kravspecifikation och utvärdering av programmeringsspråk för GENT. 1974-09-06 (FRI).

DLU 74/25 VARKAT-DLU. Ett LISP-program för simulering av ett interaktivt databassystem vid SCB. 1974-09-17. Sture Hägglund & Östen Oskarsson.

DLU 74/26 Sture Hägglund, Östen Oskarsson: Dokumentation av VARKAT-DLU. 1974-09-20.

DLU 74/27 Sture Hägglund, Östen Oskarsson: IDECS. Interactive Definition and Editing of a Conversational System or Interactive Description and evaluation of Conversation Structures. 1974-09-25.

DLU 74/28 Anders Beckman: Secondary Effects. 1974-10-03.

DLU 74/29 Torgny Tholerus: The Variable-Free Calculus, or Lambda-calculus without dummy-variables. 09-1974.

DLU 74/30 Anders Beckman, Tom Smedsaas: Experiments with Fortran preprocessors, 1974-10-08.

DLU 74/31 Anders Beckman, Tom Smedsaas: SUGFOR - Syntactical Sugar for Fortran, 1974-10-10.

DLU 74/32 Åka Malmberg: Programdokumentation NOVALISP. Korskompilator och runtimesystem till minidator Nova 1200.

DLU 74/33 K. Cohen: HASHTEST - ett program för testing av effektiviteten hos hashfunktioner och hashmetoder. 1974-11-06. (trevetygsupsats)

DLU 74/34 Lennart Beckman, Anders Haraldson, Östen Oskarsson & Erik Sandewall A partial evaluator, and its use as a programming tool. 1974-11-24.

DLU 74/35 Mats Nordström: SLISP A System for Simulation using LISP. Okt 1974.

DLU 75/1 A. Beckman, J. Wilander: Software error classification, 1975-01-11.

DLU 75/2 S. Hägglund: CICERON - A Street Guide and Information System ISBN 91-506-0007-9.

DLU 75/3 S. Hägglund, Ö. Oskarsson: IDEC2, User's Guide.

DLU 75/4 O. Willén: Filhantering och procedureditering i PLAST för DEC-10, 1975-01-21, ISBN 91-506-0006-0.

DLU 75/5 Ö. Oskarsson: About design of conversation interfaces for non-expert users. 1975-01-15.

DLU 75/6 B. Westerlund: Ändringar och förbättringar i LISP F1 och LISP F2. (trebetygsuppsats), 1975-01-07.

DLU 75/7 P. Haeggström: COMLIST: ett program för globala korsreferenslistor för commondeklarationer i Fortran. 1975-02-10. (trebetygsarbete)