STACKPOINTER

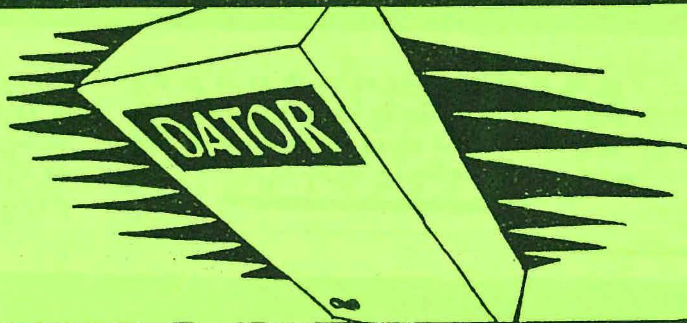
Husorgan för STACKEN – datorföreningen på KTH.
Grundad 1978

2-1982

PROLOG POWER - Stor kampanj!

SOFTNET - User Group bildas

TMS99000/Turbo



STACKPOINTER

Är organ för datorföreningen STACKEN. STACKPOINTER utkommer när material i tillräcklig mängd finns, normalt 2 — 3 gånger per år.

Ansvarig utgivare: Lars-Henrik Eriksson
Redaktör: Dag Rende
I redaktionen: Björn Danielsson
Mats Jansson
Robert Lindh
Dan Pettersson
Dag Rende

Datorföreningen STACKEN
c/o NADA, KTH
100 44 STOCKHOLM

Postgiro: 433 01 15 - 9

Medlemsskap, möten, styrelse etc. se STACKEN – info på sidan 6

INNEHÅLL

7 Nya Custom design-kretsar T.ex. en krets för röstsyntes som använder *Time Domain Linear Predictive Coded Formant Synthesis.*

8 Virka din egen LISP Del 1 av några artiklar om LISPs interna uppbyggnad.

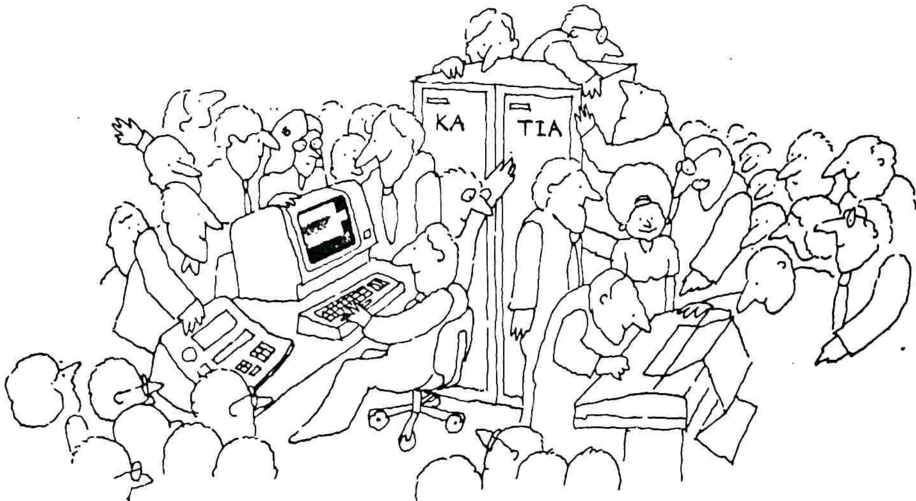
12 Sticka Lill-babs PROLOG! STACKENS PROLOG-jockey NIL monterar skivor på KSP – stationen med öppen disk-kanal.

34 TMS99000 Ny processor med gammal beprövad arkitektur.

39 Motorola 6805 Familj av enchipsprocessorer med 6-bitars SP, och 11-bitars PC!

40 SOFTNET – lägesrapport Det rör sig i det här projektet. User group håller på att bildas.

Redaktörens ruta 4
Ordföranden har ordet 5
STACKEN – **info** 6
Insändare 42



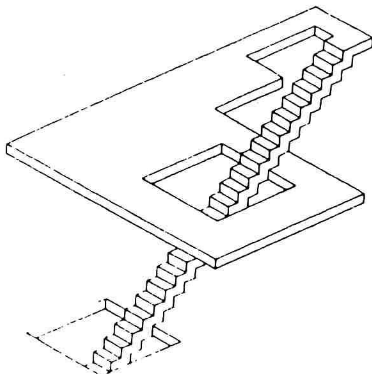
Redaktörens ruta

Ett nytt nummer, nya artiklar, nya idéer. Den här gången är det mer text och mindre bilder.

Misströsta inte om du inte förstår vad man skall ha PROLOG till. För att du skall kuna lära dig prolog måste du ju ha nån PROLOG att köra. Här är den! – Har du LISP eller PASCAL på din burk, så är det bara att knacka in. Framöver kommer det artiklar med flera exempel på hur man programmerar i PROLOG.

STACKPOINTER handlar inte bara om LISP och PROLOG. Den handlar om vad *du* skriver. Skriv! Vi önskar oss fler hårdvaruartiklar i fortsättningen.

Ser du suddigt? – Det är inte dina ögon det är fel på. Spökbilden på tecknen beror på en dåligt rengjord printer.



Ordföranden har ordet

Nu har det äntligen blivit dags att ge ut en ny STACKPOINTER. Vad har hänt sedan det senaste numret kom ut, undrar du kanske. Tja, vi har fått nya stadgar (de publicerades i förra STACKPOINTER), vi har börjat sälja dricka i 6502 igen (bara stölderna inte börjar på nytt) och vi har numera ungefär 130 medlemmar.

Efter att ha haft ganska låg aktivitet under sommaren har det nu börjat röra på sig ordentligt i föreningen. AMIS-projektet är praktiskt taget avslutat, men nya spännande saker är på gång. Det finns planer på att ge kurser och föreläsningar om Lisp, Prolog, TEX och andra saker som en högklassig hacker måste känna till. Ett nytt programmeringsprojekt ligger i startgroparna – implementering av STACKENS eget datCENSUR-CENSUR-CENSUR tem.

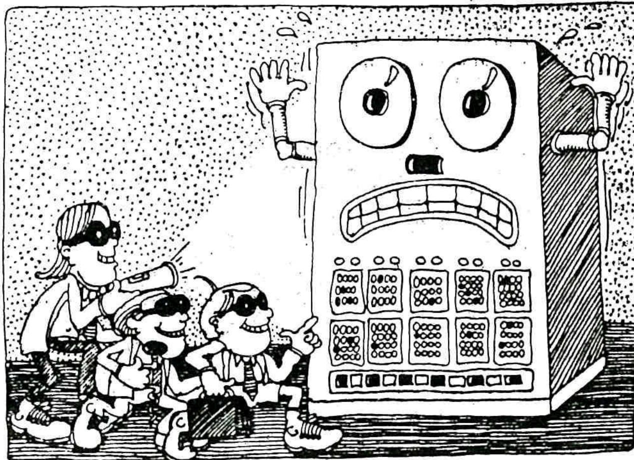
Men framför allt är det två saker som är på gång. Båda har lustigt nog mycket med hårdvara att göra. Det ena är SOFTNET. I förra numret kunde du läsa om amatörradionätet för dataöverföring. Nu börjar många i KTHs radioklubb och STACKEN planera att bygga egna SOFTNET-noder. Flera möten om SOFTNET är planerade när detta skrivs, så man kan nog räkna med att mycket kommer att hända. Snart kanske vi kan ha vårt eget SOFTNET i STACKEN.

Den andra saken gäller en dator. Flera gånger förut har det varit tal om att STACKEN skulle få köpa/låna en dator, så att vi och NADAs elever slipper trängas med varandra på Nadja, till bådas irritation. Nästan varje gång har dock något misslyckats någonstans. Denna gång har vi dock fått ett mera storslaget erbjudande än vanligt. Vi har chans att få en egen DEC-10 från Digital Equipment! När detta skrivs är det ännu inte klart hur det blir. Maskinen är STOR, den kräver massor med utrymme och elkraft, vilket vi hoppas få hjälp med ifrån Teknis. Ifall utrymme, elkraft, underhåll etc. går i lås och DEC säger definitivt ja, då står STACKEN med en egen maskin i samma klass som Nadja! Skulle det hända så har vi MASSOR av sysselsättning för mjuk- och hårdvaruintresserade medlemmar.

– Klarar Lysator av att köra stora maskiner, så kan väl vi! ∎

Lars-Henrik Eriksson

digital



STACKEN

Datorföreningen STACKEN är en partipolitiskt och religiöst obunden idéell sammanslutning, vars ändamål är att främja intresset för och vidga kunskaper inom datorområdet. Medlemskap kan beviljas efter ansökan till föreningens styrelse. (Skicka en lapp med namn och adress och vad du är intresserad av). När medlemskapet beviljats så kommer ett inbetalningskort. Medlemsavgiften är f.n. ~~70~~ kronor per år. 83:-

Första torsdagen i varje månad har STACKEN möte i någon ledig lokal på Teknis (oftast i sal E7 eller nånstans i D40-planet). Då brukar vi diskutera nya projekt, höra hur långt 16-bitsprojektet har kommit sen förra mötet, samt ägna oss åt administrativt strul. Föreningen har dessutom två årsmöten varje år, ett på våren och ett på hösten.

STACKEN har en styrelse, och den består just nu av:

Ordförande: Lars-Henrik Eriksson
Vice ordförande: Jan Michael Rynning
Sekreterare: Robert Lindh
Kassör: Evald Koitsalu
Hexmästare: Dan Norstedt
Suppleant: Hans Nordström
Redaktör: Dag Rende

Meddelanden till STACKEN kan t.ex. lämnas i brevlådan utanför lokalen (sal 508F utanför terminalsalarna i D40-planet), eller via pappersposten, med adress:

Datorföreningen STACKEN

c/o NADA, KTH

100 44 Stockholm

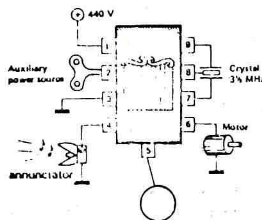
Bidrag i form av pengar eller datorer mottages gärna.

STACKENs Postgironummer är: 433 01 15 - 9

Coming soon... perhaps

New developments in consumer electronics

In recent years a number of new developments in the LSI integrated circut field have revolutionised consumer electronics. Described here are some exciting new ideas, even now nearing fruition in the engineering department of *Silicon Hollow International Technology Inc.* These new lead balloons for the 1982 Christmas season include the Cuckoo Clock chip, Smoke Dectector with snooze feature and the digital L.C.D. Sundial with frontlight and melody alarm.



CUK 100 A pinout
Figure 1.

The new nine-pin dual-in-line-and-a-bit package outline of the CUK 100A cuckoo clock chip (figure 1) allows for direct connection of the pendulum without the need for external conditioning circuitry. The audio output (pin 5) drives a piezoelectric transducer to produce an authentic "cockoo" sound encoded by *Time Domain Linear Predictive Coded Formant Synthesis.* The synthesis is based on an actual recording of a cuckoo made by the company's director of research. The high pass filter in the audio output circuit of the chip is necessary to remove a low frequency spurious signal in the cuckoo recording caused by the research director's inordinate love of baked beans. The device will be available in evaluation kit form, including chip, crystal, audio annunciator with beak and printed curcuit board.

There are three grades of kit available:

CUK 100 A EV/KIT/S - With dead sparrow
CUK 100 A EV/KIT/P - With dead pidgeon
CUK 100 A EV/KIT/C - With dead cuckoo

Silicon Hollow's projected smoke detector with snooze borrows technology from a number of areas. The smoke detector head uses the well known "*coughing canary*" system and is supplied complete with sand tray and a one year supply of bird seed. The programming of the snooze timer is taken from the company's successful microwave oven controller, with settings for RARE, MEDIUM RARE, WELL DONE and TOTALLY INCINERATED.

The digital L.C.D. sundial chip uses quadrature Hall Effect sensors to determine magnetic declination and hence latitude. By comparing local Time of day (entered via a calculator keyboard by the user) with G.M.T. (derived from a quartz crystal controlled master clock on the chip), the chip is also able to determine longitude. Knowing these factors, the chip adjusts the angle of the frontlight relative to the display, so that the shadow cast falls on an array of photoreceptors the output of which is digitised to provide the display. Thus the user receives the impression of using an authentic sundial combined with the convenience of digital L.C.D. display. ∎

Elektor December 1981

Virka din egen LISP, del 1

The great path has no gates,
Thousands of roads enter it.
When one passes through this gateless gate
He walks freely between heaven and earth.

Mumon

Ett utmärkt sätt att lära sej hur LISP fungerar är att göra ett eget LISP-system. Har man tur kan man på köpet få ett användbart programutvecklingssystem. Vem vet, du kanske kommer att använda din LISP till att skriva efterföljaren till *EMACS*, *ZORK* eller *MACSYMA*. Eller kanske för att utveckla logikprogrammeringsspråket *ALLAN*...

Den teoretiska grunden

För er som inte vet ett dugg om LISP rekommenderar jag en titt i Winston & Horn's bok *"LISP"* (Addison-Wesley 1981). För alla andra kan jag påminna om att grundbegreppen i LISP är *funktioner* och *s-uttryck*. Funktionerna är vanliga matematiska funktioner som tar argument i form av s-uttryck och lämnar ett s-uttryck som funktionsvärde. Ett s-uttryck är antingen en *atom*, t.ex. ordet *foo*, eller ett ordnat par av två s-uttryck, som skrivs så här: (*foo* . *bar*)
Ett förkortat skrivsätt används när man vill kombinera fler än två s-uttryck i en *lista*: (*Allan tar kakan*) betecknar samma sak som
(*Allan* . (*tar* . (*kakan* . *NIL*)))
Som bekant representeras också funktionsdefinitioner som s-uttryck. Det är det som gör att LISP är ett sånt häftigt programmeringsspråk.

Vad behöver man?

För att kunna köra LISP måste man hitta på ett sätt att representera s-uttryck, dom grundläggande LISP-funktionerna, och en metod att anropa funktioner som är skrivna i LISP. Hur representerar man ett s-uttryck? Ja, det mest uppenbara är att använda en textsträng som innehåller dess skrivna representation (s-uttryckets *print-name*). Och det är just vad man gör när man lagrar LISP-program och data på yttre filer. Men det är ganska korkat att använda den representationen internt. Den vanligaste metoden är att representera s-uttrycket som en pekare till ett ställe i minnet där s-uttryckets innehåll lagras. Mer om detta senare.

Dom grundläggande eller primitiva LISP-funktionerna är dom som inte själva kan skrivas i LISP. I "pure LISP" är det 5 stycken: *cons*, *car*, *cdr*, *atom* och *eq*. Till dessa tillkommer vanligtvis funktioner för in- och utmatning, aritmetiska funktioner, en mängd praktiska funktioner för felhantering, systemanrop med mera.

Slutligen behöver man en metod att anropa funktioner skrivna i LISP. Det är här *eval* och *apply* kommer in i bilden. Att jag inte tog med dom bland dom primitiva funktionerna beror på att *eval* och *apply* själva kan skrivas i LISP om man har tillgång till en alternativ metod att anropa funktioner skrivna i LISP, t.ex. med hjälp av en kompilator. Men det är mycket enklare att skriva *eval* och *apply* som primitiva funktioner. (Och sen skriva en kompilator i LISP!)

Hur ser det ut inuti?

Av dom 5 grundfunktionerna brukar *eq* vara den absolut enklaste att implementera. Argumenten, som är pekare, jämförs helt enkelt med varandra och om dom var lika lämnar man värdet *T* (förkortning för *true*), annars *NIL*. Funktionen *cons* ska ge en pekare till ett objekt som innehåller de båda argumenten. En tanke är att man då först kollar ifall det redan finns just ett sånt objekt i minnet, men det är svårt och jobbigt, så i praktiken låter man alltid *cons* reservera plats för ett nytt objekt. Exakt hur detta går till kommer jag till senare, till att börja med antar vi att man kan allokera minne sekvensiellt från en stor data-area och strunta i vad som händer när den tar slut. Cons-objektens utseende kan variera en del, det enda kravet är att dom ska innehålla två pekare. Den första lispen som man känner till från historisk tid gjordes under slutet av rör-åldern på en IBM 704. Denna maskin hade 36-bits ord och 15-bits adresser, och hade speciella instruktioner för att hantera delar av register. Delarna kallades *prefix* (3 bitar), *address* (15 bitar), *tag* (3 bitar) och *decrement* (15 bitar). 15-bitsfälten var speciellt intressanta eftersom dom kunde rymma varsin pekare. Namnen *car* och *cdr* var ursprungligen förkortningar av *C*ontents of *A*ddress part of *R*egister respektive *C*ontents of *D*ecrement part of *R*egister.

Hur lagrar man atomer då? I princip är det bara atomens egenskapslista som behöver lagras. Ibland lagrar man vissa egenskaper, t.ex. printname och funktionsdefinition, separat från egenskapslistan för att spara åtkomsttid och minne. Dessutom är det vanligt att numeriska atomer representeras så att man kan utnyttja maskinens inbyggda aritmetikinstruktioner effektivt. Men det absolut enklaste är att låta atomer vara vanliga cons-objekt där car innehåller ett speciellt värde, och cdr pekar på atomens egenskapslista. Det speciella värdet används som indikation på att objektet är en atom. Vilket värde som används är mindre viktigt, det kan t.ex. vara en pekare till atomen *ATOMHEADER*. Funktionen *atom* kan i så fall definieras:

```
(defun atom (x)
   (eq (car x)
       (quote ATOMHEADER)))
```

Ett annat, lite mer effektivt sätt att lagra atomer bygger på att man har typbittar i objekten. Ytterligare en variant är att tillämpa apartheid: atomer och listor läggs i olika delar av adressrymden. Detta har emellertid vissa nackdelar, precis som i verkligheten.

Numeriska atomer brukar av effektivitetsskäl nästan alltid lagras i binär form. Antingen genom att vissa pekare tolkas som numeriska värden istället för adresser, eller också genom att man inför speciella objekt för binärdata. Funktionen *numberp* måste då användas tillsammans med *atom* för att skilja mellan olika typer av objekt.

Utmatning av s-uttryck

Antag att vi har hittat på en lämplig intern representation för s-uttryck. Nästa steg är att skriva en funktion som översätter till den yttre representationen. Den enda intressanta frågan i det sammanhanget är hur man får fram atomers printname. För numeriska atomer är det enkelt: man skriver ut talvärdet i aktuell utmatningsbas. För symboler måste man ha print-name undanlagrat nånstans, t.ex. på egenskapslistan under indikatorn *PNAME*. Print-name kan representeras som en lista av ASCII-värden, som en lista av tal där varje tal representerar mer än ett

tecken, som ett enda tal (Gödelkodning) eller som en sträng ifall strängar finns som primitiv datatyp i lispen.

Själva print-funktionen är annars väldigt enkel i grundutförandet.

Den bör kompletteras med en pretty-printer eller något annat indenteringshjälpmedel om man ska kunna åstadkomma något vettigt med lispen.

Inläsning av s-uttryck

Inläsning av s-uttryck är en aning knepigare. Man vill att atomer som stavas likadant ska uppfylla *eq*-predikatet, d.v.s. dom ska representeras av samma pekare. Det löser man genom att spara alla atomer i en objekt-lista (vanligen kallad *oblist*) som söks igenom varje gång scannern har läst en atom. Om det fanns en atom med samma print-name i listan så används den, i annat fall skapas en ny atom och läggs in i oblistan.

Numeriska atomer brukar inte representeras unikt, och det beror på att man vill slippa söka igenom oblistan efter varje aritmetikoperation. Ofta gör man en kompromiss och låter små tal representeras unikt i alla fall, genom att reservera en bunt pekare för små tal.

Praktiskt taget alla lispar tillåter att man skriver *'foo* istället för *(quote foo)*. Tecknet ' är ett exempel på ett *readmacro*. En användbar finess är att ha en programmerbar syntaxtabell där man kan definiera egna readmacros för diverse ändamål.

EVAL och APPLY — interpretatorn

Funktionerna *eval* och *apply* utgör tillsammans själva LISP-interpretatorn. *eval* evaluerar ett s-uttryck, *apply* skickar argument till en funktion. *eval* fungerar på följande sätt: Om argumentet är ett tal, returnera talet självt. Om argumentet är en symbol, hämta motsvarande variabelvärde. Annars är argumentet en lista med funktionsnamnet som första element. Om det är ett konventionellt funktionsanrop, evaluera alla argumenten med *eval* och anropa funktionen med hjälp av *apply*. Om uttrycket är en s.k. *special form* (som *quote* eller *cond*), skicka cdr av uttrycket som parameter utan att evaluera några argument. Om uttrycket är ett makro-anrop, makroexpandera hela uttrycket och evaluera det igen. Makroexpansionen görs genom att man skickar hela uttrycket som parameter till funktionen, som lämnar tillbaka ett nytt uttryck.

För att avgöra vilken typ av anrop det är frågan om brukar man använda en indikator. *EXPR* för vanliga funktioner, *FEXPR* för special forms, och *MACRO* för makron. Indikatorn lagras lämpligtvis tillsammans med funktionsdefinitionen, antingen i form av egenskapsnamn på egenskapslistan eller ihop-consad med funktionsdefinitionen, var denna nu råkar ligga. Det naturligaste stället att lägga funktionsdefinitionen på tycker jag är samma ställe där variabelvärdena ligger. En funktion är då bara en variabel som har en funktionsdefinition som värde. Tyvärr är rätt mycket LISP-kod skriven så att den utnyttjar att vissa lispar tillåter en atom att betyda olika saker beroende på om den används som funktion eller variabel. Om man vill vara så kompatibel som möjligt bör man alltså lagra funktionsdefinitionen separat, lämpligtvis på egenskapslistan.

apply tar en funktion som första argument, och en argumentlista som andra argument. Om första argumentet är en atom så hämtar *apply* dess funktionsdefinition först. Om funktionsdefinitionen är en primitiv funktion så anropas den på något implementationsberoende sätt. Om den är ett lambda-uttryck så inleds en febril aktivitet: lambda-variablerna sätts till värdena i argumentlistan, efter att dom gamla variabelvärdena först har sparats

undan nånstans. Sedan evalueras kroppen i lambda-uttrycket, lambda-variablerna återställs och lambda-uttryckets värde returneras.

Det enda knepiga här är lambda-bindningen, d.v.s. sparandet och sättandet av lambda-variablerna. Det finns två strategier för detta, och dom kallas *deep binding* och *shallow binding*. Deep binding innebär att man kommer åt en variabels värde genom att söka i en environment-lista som innehåller alla variabler och värden. För att åstadkomma lambda-bindning skapar man en ny environment-lista där man har stoppat in lambda-variablerna och deras nya värden i början. Och för att återställa är det bara att sätta tillbaka environment-pekaren till det gamla läget. Vid deep binding brukar man låta environment-pekaren vara ett extra argument till *eval* och *apply*.

Shallow binding innebär att en variabels aktuella värde lagras hos motsvarande atom, t.ex. på egenskapslistan under indikatorn *VALUE*. För att åstadkomma lambda-bindning sparar man undan lambda-variablernas gamla värden på en stack, t.ex. processorns vanliga stack, och stoppar in dom nya värdena i atomerna. För att återställa poppar man tillbaka dom gamla värdena.

Shallow binding har fördelen att det går snabbt att gå in och ut ur en lambda-bindning, eftersom man använder snabba stack-instruktioner istället för att consa. Å andra sidan är deep binding mer kraftfullt eftersom man kan spara pekare till gamla environments och byta environment mitt i en evaluering. Med shallow binding går inte det såvida man inte har en spagetti-stack (men det ska vi inte gå in på här). Vilket man väljer beror på vilka ambitioner man har. Vill man bli färdig snabbt väljer man deep binding eftersom det är lättast att implementera. Vill man kunna köra interpreterade lisp-program fort så väljer man shallow binding. Vill man ha fullständig LISP för att kunna skriva esoteriska funktioner eller för att experimentera med lexikalisk variabel-scoping, så väljer man deep binding.

Allt detta kan verka krångligt för den oinvigde, och krånglet beror på att lambda-bindningen ska simulera vanlig matematisk variabelsubstitution (som i lambda-kalkylen — se tidigare nummer av STACKPOINTER) utan att göra en massa kopiering.

(fortsättning i nästa nummer)

Björn Danielsson

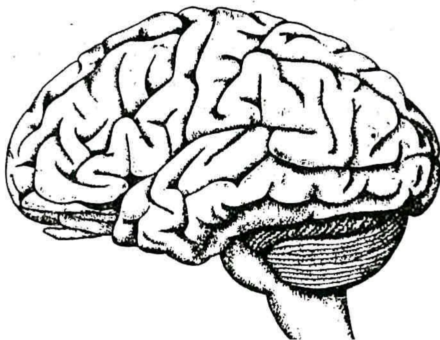
Litteratur:

[1] John Allen
"*Anatomy of LISP*" McGraw-Hill, New York 1978

[2] Guy Steele, jr
"*Data Representations in PDP-10 MacLISP*" Memo 420, Artificial Intelligence Laboratory, MIT 1977

[3] David Moon
"*MacLISP Reference Moonual*" Laboratory for Computer Science, MIT 1974

[4] Lynn Quam & Whitfield Diffie
"*Stanford LISP 1.6 Manual*" Operating Note 28.7, Stanford Artificial Intelligence Laboratory, Stanford University 1972



Sticka Lill-Babs *PROLOG*!

Hack, hack, alla bleka bittfifflare där ni kurar i källarna framför era illgröna skärmar! Stationen är KSP och NIL är din hardworking PJ och här är redan första skivan monterad /(*drunknar i "We're gonna hack - aroun' - the clock tonight"*)... och DET var Bill Hal-ey and his Kom-ets och VI är KSP och DU: PJ är inte PushJump! No, man! Inte Pop Jorn eller Poca Jola! No, no, nooo, no! *PROLOG* Jockey, man! Yeah! Tungt man! Och du, hacker, lyssna igen: Pysslat med Lisp? Testat *PROLOG*? Undrat hur det funkar? No more questions! KSP, stationen med öppen diskkanal fixar programmet för dig! Digga det här, man: Här har du en *PROLOG* skriven i Lisp! Och du, vänta, mera godis: Den är enkel! Kan köras på vilken Itsy Bitsy Machines klass Lisp som helst! Rakt! /(*"Yes, sir, I can debug" snurrar på tallriken*)/ Yeah man! Baccdra kör hårt och vi kör hårt med få kommentarer i programmet. Du: Klä på dej pjucken och STEPa och TRACEa igenom koden! Hårt? Jämän! OK, lite smulor: /(*mjuk viskande kvinnoröst med soft music i bakgrunden*) "*PROLOG*... Aahh... Så milt att du kan köra dina program baklänges..." (Beklagar sorgen att det här är på utrikiska – texten skall återanvändas.)

General Remarks

The aim of this Prolog is to show you how a typical structure-sharing Prolog interpreter works. The program is not very efficient, but once you've understood the program and the algorithm, you can easily speed them up with the facilities available on your own system. It should not be too hard even to translate the algorithm to assembler, if you like. A Pascal version of the Prolog is in an appendix to this article. To implement the Prolog in some other language you need a Lisp type symbolic expression reader and printer, and a garbage collector. The first two are simple and straightforward to write in any language. The garb is worse: If there is no such in your target language implementation, you either have to write it yourself (probably a bit cumbersome), or stay away from running large programs.

To understand this Prolog you need a some knowledge of Lisp and also some aquaintance with Prolog. The program is written in Maclisp, although only the most basic functions are used, so there should not be any big porting problems. The Lisp functions used are: APPEND, APPLY, ASSOC, ATOM, C...R, COND, CONS, DEFUN, EQUAL, GET, GO, LIST, MEMBER, NULL, PLUS, PRINC, PROG, QUOTE, READ, RETURN, SETQ, TERPRI. These functions have been chosen as a compromise in order to optimize simplicity, readability, portability, and efficiency, in that order of importance. If you don't have numbers and PLUS, you can use a simple CONS instead, but then you have to use a modified version of EQUAL in UNIFY and ASSOC. See the section on levels below. It is no big catastrophy if you don't have PROG, RETURN, and GO. They can be replaced with something WHILE-like, or by recursion, though the latter will load down the stack heavily.

Some small examples

```
                                      ; Lower case is user type-in
'<maclisp>mclisp                      ; Startup Maclisp

LISP 2122
Alloc? n                              ; n for No allocation


*

(sstatus linmode t)T                  ; Set a nicer IO-mode...

(load 'prolog)

;Loading LET 98
;Loading DEFMAX 98(WELCOME TO PROLOG!)
T
(prove (assert ((a something))       ; Programming a
               ((a else)))))          ; small example
MORE?t
NIL
(prove (a ?x) (writeln (one such x is ?x)))
(ONE SUCH X IS SOMETHING)
MORE?t                               ; With backtracking...
(ONE SUCH X IS ELSE)
MORE?t
NIL
(prove (a ?x) (writeln (one such x is ?x)))
(ONE SUCH X IS SOMETHING)
MORE?nil                             ; ... and without
T
(prove (assert                       ; A non-trivial example
        ((hello)
           (writeln (hello! what is your name?))
           (read ?x)
           (chk-answer ?x))
        ((chk-answer martin)
           (writeln (hello martin!)))
        ((chk-answer ?x)
           (writeln (i want to talk to martin!))
           (hello))))                ; Loop by recursion
MORE?t
NIL
```

```
(prove (hello))                         ; A test run
(HELLO! WHAT IS YOUR NAME?)
allan
(I WANT TO TALK TO MARTIN!)
(HELLO! WHAT IS YOUR NAME?)
martin
(HELLO MARTIN!)
MORE?nil
T
```

The corresponding in Dec-10 Prolog

```
@<prolog>prolog
Prolog-10  version 3

| ?-  assert(a(something)), assert(a(else)).

yes
| ?-  a(X), write('one such x is '), write(X), nl.
one such x is something

X = something ;
one such x is else

X = else ;

no
| ?-  a(X), write('one such x is '), write(X), nl.
one such x is something

X = something n

yes
| ?-  [user].
| hello :-
|       write('hello! what is your name?'), nl,
|       read(X),
|       chk_answer(X).
| chk_answer(martin) :-
|       write('hello martin!'), nl.
| chk_answer(X) :-
|       write('i want to talk to martin!'), nl,
|       hello.
```

```
| ↑Z
user consulted   100 words     0.49 sec.

yes
| ?-  hello.
hello! what is your name?
|: allan.
i want to talk to martin!
hello! what is your name?
|: martin.
hello martin!

yes
```

Here is a short summary of how to communicate with the Prolog: As you can see the syntax is to a large extent inherited from Lisp. The toplevel is a READ-PROVE loop. (When a Prologician says "prove a goal" he means "execute a procedure call". You type in an expression (with Lisp S-expression syntax) of the form

```
(prove <goal 1>... <goal N>)
```

where <goal> is a list (<functor> <arg 1>... <arg N>), and where <functor> is a predicate name, which is represented as a Lisp symbol. The <arg> s are general S-expressions which may contain Prolog variables, which are Lisp symbols for which the predicate IS-VAR? returns T (more about that below). Here, they are symbols with leading questionmarks. Available evaluable predicates (system functions) are ASSERT, READ, and WRITELN. ASSERT adds clauses to the database (program), a list of clauses which is pointed to by a global variable *DATABASE*. A clause is a list of goals (<goal 1>... <goal N>). The meaning of it is approximately: To prove <goal 1> try to prove <goal 2>... <goal N>. If the list only contains <goal 1>, it means that <goal 1> is true. Prolog automatically starts reading from the file INIT.PLG. During execution after all the goals in a query have been proved, you are asked the question "MORE?". If you answer NIL, it will return immediately to the toplevel. Otherwise it will backtrack and try to find alternative proofs.

Look at the examples before you start. A good way to see how the program works is to single-step through an example.

```
;;; Long Prolog Interpreter, © Martin Nilsson UPMAIL  82-11-15

(defun first (x) (car x))
(defun second (x) (cadr x))
(defun rest (x) (cdr x))
(defun functor-of (goal) (car goal))
(defun level-of (pair) (car pair))
```

```
(defun expr-of (pair) (cdr pair))
(defun current-level-goals (to-prove) (cdar to-prove))
(defun other-level-goals (to-prove) (cdr to-prove))
(defun first-goal (to-prove) (cadar to-prove))
(defun current-level (to-prove) (caar to-prove))
(defun but-first-goal (to-prove)
  (cons (cons (caar to-prove) (cddar to-prove)) (cdr to-prove)))

(defun prove fexpr (goals)
  (search-database (list (cons 0 goals)) '((bottom-of-env)) 1))

(defun search-database (to-prove env level)
  (cond ((null to-prove) (princ 'MORE?) (null (read)))
        ((null (current-level-goals to-prove))
         (search-database (other-level-goals to-prove) env level))
        ((get (functor-of (first-goal to-prove)) 'pred)
         (apply (get (functor-of (first-goal to-prove)) 'pred)
                (list to-prove env level)))
        ((prog (database assertion new-env
                rest-to-prove returned-value)
               (setq database *database*)
               (setq rest-to-prove (but-first-goal to-prove))
          loop (cond ((null database) (return nil))
                     (returned-value (return returned-value)))
               (setq assertion (first database))
               (setq database (rest database))
               (setq new-env
                     (unify (current-level to-prove)
                            (first-goal to-prove)
                            level (first assertion) env))
               (cond (new-env
                      (setq returned-value
                            (search-database
                              (cons (cons level (rest assertion))
                                    rest-to-prove)
                             new-env (plus 1 level)))))
               (go loop)))))

(defun unify (levelx x levely y env)
  (setq x (lookup (cons levelx x) env))
  (setq y (lookup (cons levely y) env))
  (cond ((equal x y) env)
        ((is-var? x) (cons (cons x y) env))
        ((is-var? y) (cons (cons y x) env))
        ((atom (expr-of x)) (cond ((equal (expr-of x) (expr-of y)) env)))
        ((atom (expr-of y)) nil)
```

```
        ((setq env (unify (level-of x) (car (expr-of x))
                          (level-of y) (car (expr-of y)) env))
         (unify (level-of x) (cdr (expr-of x))
                (level-of y) (cdr (expr-of y)) env))))

(defun lookup (pair env)
  (cond ((null (is-var? pair)) pair)
        ((further-bindings (assoc pair env) env))
        (pair)))

(defun further-bindings (bound env)
  (cond (bound (lookup (rest bound) env))))

(defun is-var? (pair)
  (member (expr-of pair) '(?a ?b ?c ?x ?y ?z ?u ?v ?w)))

;;; ---------- Some Evaluable Predicates (= System Functions)

(defun (assert pred) (to-prove env level)
  (setq *database*
        (append (instantiate (current-level to-prove)
                             (rest (first-goal to-prove)) env) *database*))
  (cond (env (search-database (but-first-goal to-prove)
                              env (plus 1 level)))))


(defun (read pred) (to-prove env level)
  (setq env (unify (current-level to-prove) (second (first-goal to-prove))
                   level (read) env))
  (cond (env (search-database (but-first-goal to-prove)
                              env (plus 1 level)))))

(defun (writeln pred) (to-prove env level)
  (princ (instantiate (current-level to-prove)
                      (second (first-goal to-prove)) env))
  (terpri)
  (search-database (but-first-goal to-prove) env level))

(defun instantiate (level x env)
  (setq x (lookup (cons level x) env))
  (cond ((atom (expr-of x)) (expr-of x))
        ((cons (instantiate (level-of x) (car (expr-of x)) env)
               (instantiate (level-of x) (cdr (expr-of x)) env)))))

;;; ---------- Setting up the database
```

```
(setq *database* nil)

(load 'init/.plg)
```

The Lisp variable ENV (for ENVironment) is an A-list holding Prolog variable bindings. The format of a binding is

```
((<variable-level> . <variable-name>)
 (<expr-level> . <expr>))
```

which means that the variable `<variable-name>` is bound to `<expr>`. You might wonder what all the levels and what the Lisp variable LEVEL are for. They are needed to differ variables with the same name but from different proof levels, like local variables on different block levels in Algol. Here, levels are implemented as numbers, but that is not necessary. The only thing that is needed is to create unique identifying pointers which can be compared by EQUAL in UNIFY and ASSOC. It is possible to use special version of EQUAL in them so that you can replace the PLUS with a CONS, or simply the new value of the variable TO-PROVE.

TO-PROVE is the most important Lisp variable. It is a list of continuations, or goals left to prove. Each element has the format

```
(<level> <goal 1> ... <goal n>)
```

In principle, the interpreter has only two parts: The SEARCH-DATABASE function which searches the database, ie the program, for clauses (assertions) whose heads match the goals to prove. The other is UNIFY, which is a two-way pattern matcher.

Some comments to the program:

Lines `(defun first...` - `(cons (cons...` :
Selector and constructor functions. Only for readability. Suitable to implement as macros if you have such.

Lines `(defun prove...` - `(search-database...` :
Setting up initial args.

Line `(cond ((null to-prove)...` :
If null, everything is proved and we are finished. Time to ask the user if he wants to backtrack.

Lines `((null (current-level-goals...` - `(search-database...` :
If null, everything is proved on this level, so go on to next.

Lines `((get...` - `(list ...` :
Check if this goal is a call to an evaluable predicate. They are functions which have their function properties under the indicator PRED on the property lists. In Maclisp they can be defined by

```
(defun (<fun> pred) <arglist> <body>)
```

but you can also do

```
(putprop 'pred '(lambda <arglist> <body>) '<fun>)
```

Line `(prog...` :
Here we start searching for something in the database matching our goal.

Line `(setq database...` :
The global variable *database* contains the database.

Line `(returned-value...` :
If a recursive call to SEARCH-DATABASE returned a value (see below), then that value should drop through and be returned.

Lines `(setq new-env...` - `level...` :
UNIFY is called with five args

```
(UNIFY <levelx> <x> <levely> <y> <env>)
```

If <x> and <y> don't match, UNIFY returns NIL. Otherwise, it returns a new enviroment, which is the old one with possible new bindings pushed on top of it. For instance

```
(UNIFY '17 '(A ?X) '4711 '(?Y B) '(((3 . ?Z) . (2 . C))
                                    ((BOTTOM-OF-ENV)))) =
= (((4711 . ?Y) . (17 . A))
   ((17 . ?X) . (4711 . B))
   ((3 . ?Z) . (2 . C))
   ((BOTTOM-OF-ENV)))
```

Lines `(cond...` - `new-env...` :
If UNIFY succeeded we add the tail of the chosen assertion which which possibly contains some new goals to prove and call SEARCH-DATABASE recursively.

Line `(go loop)` :
Backtrack! Sometimes we want to stop backtracking (like CUT in Dec-10 Prolog) which can be done by returning a non-nil value from SEARCH-DATABASE. Note that this feature is used in the toplevel function bound to the global variable *TOPFN: If the user answers the "MORE"-question with NIL, the value T is returned, and falls through all the way down to PROVE.

Lines `(defun unify...` - `(setq y...` :
The first thing to do is to lookup Prolog variable values in the A-list ENV. What we

get back are pairs `(level . expr)`. If a variable is unbound, `LOOKUP` just returns what it got as its first arg.

Lines `((is-var? x...` - `((is-var? y...` :
If one of `x` and `y` was an unbound variable, then bind it.

Lines `((atom (expr-of x...` - `((atom (expr-of y...` :
If one of `x` and `y` was a Prolog constant (Lisp atom), the other one must be equal to it.

Lines `((setq env...` - `(unify...` :
Otherwise, both are cons-cells, and `UNIFY` is first called on their cars, and then on their cdrs.

Lines `(defun lookup...` - `(pair...` :
Note that a variable can be bound to another variable, which in its turn might be bound to another variable, and so on. `ASSOC` returns `NIL` if it can't find anything.

Lines `(defun is-var?...` - `(member...` :
A better test would be to check first if it is a symbol, and then whether the first char of the printname is a questionmark. You can change the syntax of variables by changing this function.

If you wonder about the call to `INSTANTIATE` in the evaluable predicate `WRITE`: In the expression to be written there might be variables with values that have to be looked up before printing. `INSTANTIATE` looks up all bindings and substitutes them into the expression to be printed.

(*tonar ut*)/ och det säger KSP som lägger på mera macho muzak med mungiga Kraftwerk/(*otryckbart*)/ Petat in *PROLOG*en ännu? Polarn: ett nafsverk! Haft värk i känselspröten? COBOL-fingrar? DÅ har VI den allra SENASTE *PROLOG*en för DIG: 18 rader! Begrunda! Funkar som gamla! Mindre! Snabbare! Oläsligare! Bättre på alla sätt! Och hacker, var beredd: Din logikwizard skickar nya hotta evaluerbara predikat! Spana in el ejemplos och TESTA! Följande gäller: `CUT` är gömt i IF! Det är DET som gäller, man! HACKA! Det är DET som är Saturday Night Fever – och här kommer jingeln /(*rytmiskt dunkande växer fram*)...

```
;;; Short Prolog interpreter, © Martin Nilsson UPMAIL  82-11-15

(defun prove fexpr (c) (pr (list (cons 0 c)) '((bottom-of-env)) 1))

(defun pr (c e n)
 (let* ((f (cadar c)) (d (and f (cons (cons (caar c) (cddar c)) (cdr c)))))
  (cond
    ((null (cdar c)) (cond (c (pr (cdr c) e n)) ((funcall *topfn* n e))))
    ((and (null (atom f)) (setq f (get (car f) 'pred))) (funcall f c e n))
    ((do ((b *database* (cdr b))) ((or (null b) f) f)
   (and (setq f (unify (caar c) (cadar c) n (caar b) e))
(setq f (pr (cons (cons n (cdar b)) d) f (add1 n)))))))))
```

```
(defun unify (m x n y e)
 (cond ((equal (setq x (ult (cons m x) e)) (setq y (ult (cons n y) e))) e)
       ((cond ((var x) (cons (cons x y) e)) ((var y) (cons (cons y x) e))))
       ((or (atom (cdr x)) (atom (cdr y))) (and (eq (cdr x) (cdr y)) e))
       ((setq e (unify (car x) (cadr x) (car y) (cadr y) e))
        (unify (car x) (cddr x) (car y) (cddr y) e))))

(defun ult (p e) (or (and p (var p) (ult (cdr (assoc p e)) e)) p))

(defun var (x) (and (symbolp (cdr x)) (eq (getchar (cdr x) 1) '?)))

;;; ---------- Evaluable Predicates

(defun restof (c) (cons (cons (caar c) (cddar c)) (cdr c)))

(defun narg (n c e) (narg1 n (cons (caar c) (cadar c)) e))

(defun narg1 (n p e)
 (cond ((progn (setq p (ult p e)) (zerop n)) (list (car p) (cadr p)))
       ((narg1 (sub1 n) (cons (car p) (cddr p)) e))))

(defun (lisp pred) (c e n)
 (let ((a1 (narg 1 c e)) (a2 (narg 2 c e)))
  (setq e (unify (car a2) (cadr a2) n (eval (inst (car a1)
                                                  (cadr a1) e)) e))
  (and e (pr (restof c) e (1+ n)))))

(defun inst (n x e)
 (cond ((atom (cdr (setq x (ult (cons n x) e)))) (cdr x))
       ((cons (inst (car x) (cadr x) e) (inst (car x) (cddr x) e)))))

(defun (if pred) (c e n)
 (let ((ne (let ((*topfn* 'cons)) (pr (list (narg 1 c e)) e n))))
   (cond (ne (pr (cons (narg 2 c e) (cdr c)) (cdr ne) (car ne)))
         ((pr (cons (narg 3 c e) (restof c)) e n)))))

(defun setof-topfn (n e) (push (inst (car *x*) (cadr *x*) e) *l*) nil).

(defun (setof pred) (c e n)
 (let ((*x* (narg 1 c e)) (*l*) (*topfn* 'setof-topfn) (a3 (narg 3 c e)))
   (pr (list (narg 2 c e)) e n) (setq e (unify (car a3) (cadr a3) n *l* e)))
 (and e (pr (restof c) e (1+ n))))

(defun (call pred) (c e n)
 (let* ((x (narg 1 c e)) (y (ult (cons (car x) (cadr x)) e)))
  (setq x (cons (cdr (ult (cons (car y) (cadr y)) e)) (cddr y)))
```

```
  (pr (cons (list (car y) x) (restof c)) e n)))

;;; ---------- Setting up the data base

(setq *topfn* '(lambda (c e) (princ 'MORE?) (null (read)))
      *database*
      '(((assert . ?x) (lisp (setq *database* (append  ?x *database*)) ?y)))
      *infile* nil
      *outfile* nil)

(load 'init/.plg)
```

Comments to the short Prolog

This Prolog contains the previous Prolog as a subset. Some differences are: The MacLisp functions LET, LET*, DO, and FUNCALL are used. They can however all be replaced by the previously mentioned functions according to the following patterns:

```
(LET ((<var1> <expr1>)... (<varN> <exprN>)) <body>)
=> ((LAMBDA (<var1>... <varN>) <body>) <expr1>... <exprN>)

(LET* ((<var1> <expr1>)... (<varN> <exprN>)) <body>)
=> (LET ((<var1> <expr1>))
     (LET ((<var2> <expr2>))
      ...
      (LET ((<varN> <exprN>)) <body>)... ))

(DO ((<var> <start> <incr>)) (<cond> <value>) <body>)
=> (PROG (<var>)
         (SETQ <var> <start>)
   LOOP  (COND (<cond> (RETURN <value>)))
         <body>
         (GO LOOP))

(FUNCALL <f> <arg1>... <argN>)
=> (APPLY <f> (LIST <arg1>... <argN>))
```

In the function ULT and the function PR, in the two lines " (LET*..." and " (COND..." it is assumed that CAR and CDR of NIL is NIL, and that AND and OR returns the value of their last evaluated argument.

INIT.PLG

```
(prove (assert
   ((t))
   ((== ?x ?y) (lisp (equal '?x '?y) t))
   ((= ?x ?x))
   ((cannot-prove ?x) (if (call ?x) (nil) (t)))
   ((not= ?x ?y) (cannot-prove (= ?x ?y)))
   ((read ?x) (lisp (read *infile*) ?x))
   ((write ?x) (lisp (princ '?x *outfile*) ?y))
   ((nl) (lisp (terpri *outfile*) ?x))
   ((writeln ?x) (write ?x) (nl))
   ((tell ?file)
    (lisp (and *outfile* (close *outfile*)) ?y)
    (lisp (setq *outfile* (and '?file (open '?file 'out))) ?x))
   ((see ?file)
    (lisp (and *infile* (close *infile*)) ?y)
    (lisp (setq *infile* (and '?file (open '?file 'in))) ?x))
   ((retract ?x)
    (lisp *database* ?db1)
    (remove ?x ?db1 ?db2)
    (lisp (setq *database* '?db2) ?y))
   ((remove ?x (?x . ?1) ?1))
   ((remove ?x (?y . ?11) (?y . ?12)) (remove ?x ?11 ?12))
   ((append nil ?x ?x))
   ((append (?x . ?y) ?z (?x . ?u)) (append ?y ?z ?u))
   ((prettyprint ?x)
    (lisp (funcall (or prini 'prini) '?x *outfile*) ?y) (nl) (nl))
   ((member ?x (?x . ?y)))
   ((member ?x (?a . ?y)) (member ?x ?y))
   ((var ?x) (lisp (var '(() . ?x)) t))
   ((length ?1 ?n) (lisp (length '?1) ?n))
   ((name ?x ?y)
    (if (var ?y) (lisp (exploden '?x) ?y) (lisp (implode '?y) ?x))))

      (writeln (welcome to prolog!))
      (nil))
```

A new feature is the escape-to-Lisp predicate LISP called by (LISP ?X ?Y). ?X is evaluated by the Lisp interpreter and the result is unified with ?Y. LISP is very handy when one implements things like READ and WRITE. Look at the database for examples of use! This Prolog also has a simple kind of filehandling. If an expression ?X is constructed, it can subsequently be interpreted by (CALL ?X). The IF and SETOF predicates are common extensions to Pure Prolog. They might be somewhat

esoteric for the unexperienced Prolog user, so we shall not go into too deep a detail. (IF ?X ?Y ?Z) can be paraphrased as "if ?X then ?Y else ?Z" and (SETOF ?X ?Y ?Z) as "?Z is a list of all ?X such that ?Y". If in Dec-10 Prolog would be

```
if(X,Y,Z) :- (call(X),!,call(Y)); call(Z).
```

SETOF is like Dec-10 Prolog's setof, but does automatic existential quantification of all free variables in ?Y, and it does not fail if an empty list is generated (don't worry if this doesn't tell you anything).

Comments to the Pascal version

This is a direct translation of the first Lisp Prolog to Pascal. Except for underscore characters in identifiers, the program conforms strictly to Pascal as defined in the Pascal Report by Jensen and Wirth. The program has a Lisp type symbolic expression reader and printer, and explicit versions of CAR, CDR, CONS and and a few other standard Lisp functions. Note that the reader does *not* convert lowercase to uppercase. The names of the evaluable predicates are in lower case. Unfortunately, there is no requirement of Pascal to have a garbage collector, something which we would have appreciated. However, if your Pascal has MARK and RELEASE, RELEASE can be called each time the interpreter returns to the toplevel. The heap can be cleared from everything allocated after the most recent allocation of a cell in the structure GLOBAL_DATABASE.

(*Sunday Morning Fever fade out*)/ och det ryktas förresten att Tra-volta hade V24 i snitt när han slutade plug-get. Började han köra buss sedan, tro? Bättre som busschaffisar passar killarna L. Kristiansson och A. Karsberg! Men deppa inte över dom, man! Flukta in Japsen! Femte generationen: Logic is the word! Jäpp – vad finns i första datalabbkursen på MIT? You guessed it! En *PROLOG* i Scheme! Och ska VI gissa på nästa låt... Put another nickel in? Basic, Basic, Basic?? BC??? A wingless bird with hairy feathers???? No way, man! ...*PROLOG* rock! Stort! /(*dånar*)/ Och du, natthackare, uret är miljoner och vi på KSP rasar vidare mot nästa sändning: Compiler för *PROLOG*! Väg det, man! G'natt från NIL, Sayoonara, Dosvedanja, en tanke till stilbildar'n Berggren, CUAGN! The handle is KSP, and I'm pullin' the plug! Fortyseven eleven is clear! ∎

Martin NILsson

Filer med programmen i artikeln finns på datorn VERA. Filen <aida.martin-nilsson>prolog.txt talar om var de olika filerna ligger.

Appendix, PROLOG i Pascal

```
PROGRAM prolog(input,output,init);

{ Prolog Interpreter, © Martin Nilsson UPMAIL 82-11-15

Syntax:          Lisp S-expressions
Toplevel:        READ-PROVE loop
Input:           (prove <goal 1>... <goal N>)
<goal>:          (<functor> <arg 1>... <arg N>)
<functor>:       a predicate name, which is represented as a Lisp symbol
<arg>:           S-expressions in general which may contain variables
<variable>:      Lisp symbol with a leading questionmark
Evaluable predicates: assert, read, writeln
(assert <clause 1>... <clause N>): adds clauses to the database
<clause>:        (<goal 1>... <goal N>). Paraphrased: To prove <goal 1> try to
                 prove <goal 2>... <goal N>.
(read ?x):       Read an expression and unify with ?x
(writeln ?x):    Write ?x and a new line
Initialization: Starts reading from an INIT file.
Backtracking:    During execution after all the goals in a query have been
                 proved, you are asked the question "MORE?". If you answer
                 NIL, it will return immediately to the toplevel. Otherwise
                 it will backtrack and try to find alternative proofs.       }

LABEL 1 {error return};

CONST txtlength = 10;
      empty = ';'; tab = '       ';
      emptypname      = ';;;;;;;;;;';
      t_pname         = 't;;;;;;;;;';
      nil_pname       = 'nil;;;;;;;';
      b_of_env_pname  = 'b_of_env;;';
      prove_pname     = 'prove;;;;;';
      read_pname      = 'read;;;;;;';
      writeln_pname   = 'writeln;;;';
      assert_pname    = 'assert;;;;';

TYPE txt = PACKED ARRAY [1..txtlength] OF Char;
     celltype = (pair,symbol,number);
     ptr = ↑cell;
     cell = RECORD
            CASE ptype : celltype of
```

```
                pair : (car : ptr; cdr : ptr);
                symbol : (pname : txt);
                number : (value : integer);
            END;

VAR bufferchar : Char; endline : Boolean; init : Text;
    global_database,expr,nill,temp : ptr;
{   for_gc_only_if_you_have_mark_and_release : Integer; }

{ ---------- Lisp Structure Funs }

PROCEDURE error(msg : txt); BEGIN Writeln(msg); GOTO 1; END;

FUNCTION car(x : ptr) : ptr; BEGIN car:=x↑.car END;
FUNCTION cdr(x : ptr) : ptr; BEGIN cdr:=x↑.cdr END;

FUNCTION cons(x,y : ptr) : ptr;
    VAR p : ptr;
BEGIN new(p); p↑.ptype:=pair; p↑.car:=x; p↑.cdr:=y; cons:=p END;

FUNCTION symbolcons(x : txt) : ptr;
    VAR p : ptr;
BEGIN new(p); p↑.ptype:=symbol; p↑.pname:=x; symbolcons:=p END;

FUNCTION numbercons(x : Integer) : ptr;
    VAR p : ptr;
BEGIN new(p); p↑.ptype:=number; p↑.value:=x; numbercons:=p END;

FUNCTION equal(x,y : ptr) : Boolean;
BEGIN IF x=y THEN equal:=True
    ELSE IF x↑.ptype<>y↑.ptype THEN equal:=False
    ELSE CASE x↑.ptype OF
        pair : IF Not equal(car(x),car(y)) THEN equal:=False
            ELSE equal:=equal(cdr(x),cdr(y));
        symbol : equal:=(x↑.pname=y↑.pname);
        number : equal:=(x↑.value=y↑.value);
    END;
END;

FUNCTION atom(x : ptr) : Boolean; BEGIN atom:=(x↑.ptype<>pair) END;

FUNCTION assoc(x,l : ptr) : ptr;
BEGIN IF l=nill THEN assoc:=nill
    ELSE IF equal(x,car(car(l))) THEN assoc:=car(l)
    ELSE assoc:=assoc(x,cdr(l));
END;
```

```
FUNCTION append(x,y : ptr) : ptr;
BEGIN IF x=nill THEN append:=y
    ELSE append:=cons(car(x),append(cdr(x),y));
END;

{ ---------- Lisp Syntax Reader }

PROCEDURE myread(VAR ch : Char);
    BEGIN IF Eof(init) THEN read(ch) ELSE read(init,ch) END;
PROCEDURE myreadln; BEGIN IF Eof(init) THEN readln ELSE readln(init) END;
FUNCTION myeoln : Boolean;
    BEGIN IF Eof(init) THEN myeoln:=eoln ELSE myeoln:=eoln(init) END;

FUNCTION readch : Char;
    VAR ch : Char;
BEGIN IF bufferchar<>empty THEN BEGIN readch:=bufferchar; bufferchar:=empty;
    END ELSE IF endline THEN BEGIN myreadln; endline:=False; readch:=readch;
    END ELSE IF myeoln THEN BEGIN endline:=True; readch:=' ';
    END ELSE BEGIN myread(ch); readch:=ch END;
END;

PROCEDURE ratom(VAR atom : ptr; VAR breakchar : Char);
    VAR ch : Char; i : Integer; pname : txt;
BEGIN pname:=emptypname;
    ch:=readch; WHILE ch=' ' DO ch:=readch; breakchar:=ch;
    IF Not (ch IN ['(',')','.']) THEN BEGIN i:=0;
        REPEAT IF i<txtlength THEN BEGIN i:=i+1; pname[i]:=ch END;
            ch:=readch;
        UNTIL ch IN [' ',tab,'(',')','.'];
        IF (i=3) AND (pname[1]='n') AND (pname[2]='i')
            AND (pname[3]='l') THEN atom:=nill
        ELSE atom:=symbolcons(pname);
        bufferchar:=ch;
    END;
END;

FUNCTION readtail(firstelem : Boolean) : ptr; FORWARD;

FUNCTION lread : ptr;
    VAR atom : ptr; breakchar : Char;
BEGIN ratom(atom,breakchar);
    IF breakchar='(' THEN lread:=readtail(True)
    ELSE IF (breakchar=')') OR (breakchar='.') THEN error('Syntax R!!')
    ELSE lread:=atom;
END;
```

```
FUNCTION readtail{(firstelem : Boolean) : ptr};
    VAR atom : ptr; breakchar : Char;
BEGIN ratom(atom,breakchar);
    IF breakchar='(' THEN readtail:=cons(readtail(True),readtail(False))
    ELSE IF breakchar=')' THEN readtail:=nill
    ELSE IF (breakchar='.') AND firstelem THEN error('Syntax R!!')
    ELSE IF breakchar='.' THEN BEGIN readtail:=lread;
        ratom(atom,breakchar);
        IF Not (breakchar=')') THEN error('Syntax R!!');
    END ELSE readtail:=cons(atom,readtail(False));
END;

{ ---------- Lisp Syntax Printer }

PROCEDURE printtail(expr : ptr); FORWARD;

PROCEDURE lprint(expr : ptr);
    VAR i : Integer;
BEGIN IF expr=nill THEN write('nil') ELSE
      CASE expr↑.ptype OF
          pair : BEGIN write('('); printtail(expr) END;
          symbol : FOR i:=1 TO txtlength DO
              WITH expr↑ DO IF pname[i]<>empty THEN write(pname[i]);
          number : write(expr↑.value);
      END;
END;

PROCEDURE printtail{(expr : ptr)};
BEGIN IF expr=nill THEN write(')')
      ELSE IF expr↑.ptype=pair THEN WITH expr↑ DO BEGIN
          lprint(car); IF cdr<>nill THEN write(' '); printtail(cdr);
      END ELSE BEGIN write('. '); lprint(expr); write(')'); END;
END;

{ ---------- Prolog Kernel }

FUNCTION first(x : ptr) : ptr; BEGIN first:=car(x) END;
FUNCTION rest(x : ptr) : ptr; BEGIN rest:=cdr(x) END;
FUNCTION level_of(x : ptr) : ptr; BEGIN level_of:=car(x) END;
FUNCTION expr_of(x : ptr) : ptr; BEGIN expr_of:=cdr(x) END;
FUNCTION second(x : ptr) : ptr; BEGIN second:=car(cdr(x)) END;
FUNCTION functor_of (goal : ptr) : ptr; BEGIN functor_of:=first(goal) END;
FUNCTION goals_of_current_level (to_prove : ptr) : ptr;
   BEGIN goals_of_current_level:=cdr(car(to_prove)) END;
FUNCTION other_level_goals (to_prove : ptr) : ptr;
```

```
    BEGIN other_level_goals:=cdr(to_prove) END;
FUNCTION first_goal (to_prove : ptr) : ptr;
    BEGIN first_goal:=car(cdr(car(to_prove))) END;
FUNCTION current_level (to_prove : ptr) : ptr;
    BEGIN current_level:=car(car(to_prove)) END;
FUNCTION but_first_goal (to_prove : ptr) : ptr;
    BEGIN but_first_goal:=cons(cons(car(car(to_prove)),
        cdr(cdr(car(to_prove)))),cdr(to_prove));
    END;

FUNCTION prove(goals : ptr) : ptr; FORWARD;
FUNCTION search_database(to_prove,env,level : ptr) : ptr; FORWARD;
FUNCTION unify(levelx,x,levely,y,env : ptr) : ptr; FORWARD;
FUNCTION lookup(x,env : ptr) : ptr; FORWARD;
FUNCTION is_var(x : ptr) : Boolean; FORWARD;
FUNCTION is_eval_predicate(functor : ptr) : Boolean; FORWARD;
FUNCTION apply(functor,to_prove,env,level : ptr) : ptr; FORWARD;
FUNCTION further_bindings(bound,env : ptr) : ptr; FORWARD;

FUNCTION prove{(goals : ptr) : ptr};
BEGIN prove:=search_database(cons(cons(numbercons(0),goals),nill),
    cons(cons(symbolcons(b_of_env_pname),nill),nill),numbercons(1));
END;

FUNCTION search_database{(to_prove,env,level : ptr) : ptr};
    VAR database,assertion,new_env,rest_to_prove,returned_value : ptr;
BEGIN IF to_prove=nill THEN BEGIN write('MORE? ');
        IF lread=nill THEN search_database:=symbolcons(t_pname)
        ELSE search_database:=nill;
    END ELSE IF goals_of_current_level(to_prove)=nill THEN search_database:=
        search_database(other_level_goals(to_prove),env,level)
    ELSE IF is_eval_predicate(functor_of(first_goal(to_prove))) THEN
        search_database:=
            apply(functor_of(first_goal(to_prove)),to_prove,env,level)
    ELSE BEGIN
        database:=global_database;
        rest_to_prove:=but_first_goal(to_prove);
        returned_value:=nill;
        WHILE (database<>nill) AND (returned_value=nill) DO BEGIN
            assertion:=first(database);
            database:=rest(database);
            new_env:=unify(current_level(to_prove),first_goal(to_prove),
                          level,first(assertion),env);
            IF new_env<>nill THEN returned_value:=
                search_database(cons(cons(level,rest(assertion)),
                    rest_to_prove),new_env,numbercons(level↑.value+1));
```

```
        END;
        search_database:=returned_value;
    END;
END;

FUNCTION unify{(levelx,x,levely,y,env : ptr) : ptr};
BEGIN
    x:=lookup(cons(levelx,x),env);
    y:=lookup(cons(levely,y),env);
    IF equal(x,y) THEN unify:=env
    ELSE IF is_var(x) THEN unify:=cons(cons(x,y),env)
    ELSE IF is_var(y) THEN unify:=cons(cons(y,x),env)
    ELSE IF atom(expr_of(x)) OR atom(expr_of(y)) THEN BEGIN
        IF equal(expr_of(x),expr_of(y)) THEN unify:=env ELSE unify:=nill;
    END ELSE BEGIN env:=unify(level_of(x),car(expr_of(x)),
                              level_of(y),car(expr_of(y)),env);
        IF env=nill THEN unify:=nill
        ELSE unify:=unify(level_of(x),cdr(expr_of(x)),
                          level_of(y),cdr(expr_of(y)),env);
    END;
END;

FUNCTION lookup{(x,env : ptr) : ptr};
   VAR temp : ptr;
BEGIN IF Not(is_var(x)) THEN lookup:=x
    ELSE BEGIN temp:=further_bindings(assoc(x,env),env);
        IF temp<>nill THEN lookup:=temp ELSE lookup:=x;
    END;
END;

FUNCTION further_bindings{(bound,env : ptr) : ptr};
BEGIN IF bound<>nill THEN further_bindings:=lookup(cdr(bound),env)
    ELSE further_bindings:=nill;
END;

FUNCTION is_var{(x : ptr) : Boolean};
    VAR temp : ptr;
BEGIN temp:=cdr(x);
    IF temp↑.ptype<>symbol THEN is_var:=False
    ELSE is_var:=(temp↑.pname[1]='?');
END;

{ ---------- Some Evaluable Predicates (= System Functions) }

FUNCTION instantiate(level,x,env : ptr) : ptr;
BEGIN x:=lookup(cons(level,x),env);
```

```
    IF atom(cdr(x)) THEN instantiate:=cdr(x)
    ELSE instantiate:=cons(instantiate(car(x),car(cdr(x)),env),
                              instantiate(car(x),cdr(cdr(x)),env));
END;

FUNCTION pread(to_prove,env,level : ptr) : ptr;
BEGIN env:=unify(current_level(to_prove),
        second(first_goal(to_prove)),level,lread,env);
    IF env<>nill THEN pread:=
        search_database(but_first_goal(to_prove),env,
            numbercons(level↑.value+1))
    ELSE pread:=nill;
END;

FUNCTION pwrite(to_prove,env,level : ptr) : ptr;
BEGIN lprint(instantiate(current_level(to_prove),
                            second(first_goal(to_prove)),env));
    writeln;
    pwrite:=search_database(but_first_goal(to_prove),env,level);
END;

FUNCTION assert(to_prove,env,level : ptr) : ptr;
BEGIN global_database:=append(instantiate(current_level(to_prove),
        rest(first_goal(to_prove)),env), global_database);
{   mark(for_gc_only_if_you_have_mark_and_release); }
    assert:=search_database(but_first_goal(to_prove),env,level);
END;

FUNCTION is_eval_predicate{(functor : ptr) : Boolean};
BEGIN WITH functor↑ DO is_eval_predicate:=
    ((pname=read_pname) or (pname=writeln_pname) or (pname=assert_pname));
END;

FUNCTION apply{(functor,to_prove,env,level : ptr) : ptr};
BEGIN IF functor↑.pname=read_pname THEN apply:=pread(to_prove,env,level)
    ELSE IF functor↑.pname=writeln_pname THEN apply:=pwrite(to_prove,env,level)
    ELSE IF functor↑.pname=assert_pname THEN apply:=assert(to_prove,env,level);
END;

{ ---------- Main Loop }

BEGIN
    Reset(init); endline:=False; bufferchar:=empty;
    nill:=symbolcons(nil_pname);
    global_database:=nill;
{   mark(for_gc_only_if_you_have_mark_and_release); }
```

```
 1:  WHILE True DO BEGIN write('Prolog> ');
          expr:=lread;
          temp:=car(expr);
          IF temp↑.pname=prove_pname THEN temp:=prove(cdr(expr))
             ELSE BEGIN lprint(temp); error('Input R!!!'); END;
 {        release(for_gc_only_if_you_have_mark_and_release); }
          writeln;
     END;
 END.
```

Appendix, Körexempel i Pascal PROLOGen

```
@execute prolog.pas
LINK:   Loading
[LNKXCT PROLOG execution]
INPUT       :
OUTPUT      :
INIT        : init.plg
[INPUT, end with ↑Z: ]
                                        ; Pascal wants a CRLF here
Prolog> (welcome to prolog!)

Prolog> (prove (assert ((a something))
                       ((a else)))
MORE? t

Prolog> (prove (a ?x) (writeln (one such x is ?x)))
(one such x is something)
MORE? t
(one such x is else)
MORE? t

Prolog> (prove (a ?x) (writeln (one such x is ?x)))
(one such x is something)
MORE? nil
```

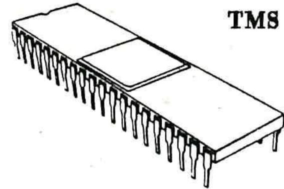
```
Prolog> (prove (assert
        ((hello)
           (writeln (hello! what is your name?))
           (read ?x)
           (chk-answer ?x))
        ((chk-answer martin)
           (writeln (hello martin!)))
        ((chk-answer ?x)
           (writeln (i want to talk to martin!))
           (hello))))
MORE? t

Prolog> (prove (hello))
(hello! what is your name?)
allan
(i want to talk to martin!)
(hello! what is your name?)
martin
(hello martin!)
MORE? nil

Prolog>
```

Texas Instruments TMS 99000

Följande artikel är huvudsakligen baserad på innehållet i den preliminära produkt-specifikationen för TMS 99000. Utgiven av Texas Instruments i November 1981. Figurer och tabeller kommer även de ifrån TI.



TMS 99000

TMS 99000, Vad är det?

TMS 99000 är Texas Instruments senaste 16-bitars processor. Den är en kraftig förbättring av TMS 9900. Det är inte så konstigt m.a.p. den snabba utveckling som skett sedan mitten på 70-talet. När företag som Motorola och Zilog gav sig in i leken med 16-bitars processorer, blev TMS 9900 hopplöst föråldrad. Med TMS 99000 så har TI presenterat sin andra generation 16-bitars processorer. För TMS 99000 är inte namnet på en processor utan en familj av processorer, se Fig. 1, på samma sätt som ZILOG Z8000 är ett familjenamn.

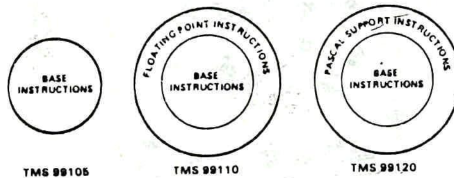


Fig. 1

TI har låtit TMS 99000 behålla den grundläggande instruktionsuppsättningen från TMS 9900. Detta var inte oväntat då TI har den uppsättningen på alla processorer i 99-serien. TMS 99000 är därför den 16-bitars processor som haft störst utbud av mjukvara när den har presenterats.

TMS 99000 vs. TMS 9900

Skillnaden mellan TMS 99000 och TMS 9900 är vid första anblicken inte så stor. TMS 99000 har fått ett antal nya instruktioner, dels instruktioner som funnit på processorer som kommit senare än TMS 9900. Men även helt nya instruktioner.

När man tittar lite närmre på TMS 99000 så upptäcker man att det finns en hel del nytt i den. En viktig sak är att all I/O inte längre är seriell, vilket var fallet med TMS 9900. TMS 99000 har fått 32 K I/O-adresser, TMS 9900 hade 4 K. De övre 16 K adresserna är för parallell I/O, byte eller ord. Men endast en liten del av adressrymden är tillgänglig för användare utan priviligier, se Fig 2.

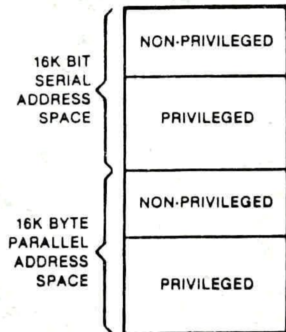


Fig. 2

Design à la TI

TMS 99000 har något som heter *MACROSTORE*, det är ett ROM som innehåller emulering av odefinierade instruktioner, plus en del annat som jag skall nämna senare. **TMS 99105** saknar MACROSTORE men man kan skapa sin egen och placera utanför processorn. **TMS 99110** innehåller flyttalsaritmetik och **TMS 99120** innehåller delar av Runtime-Systemet till TI:s Pascal. En inte alltför vild gissning är att det troligen kommer en motsvarande version för Ada, som TI lär arbeta med. Fig. 3 visar hur minnet i MACROSTORE används.

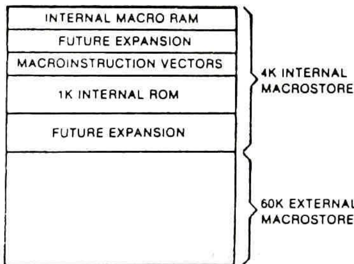


Fig. 3

TMS 99000 har en s.k. minne-till-minne-arkitektur. Det innebär att det inte finns några ackumulatorer i processorn, utan allt ligger i RAM utanför. Detta har både för- och nackdelar, men det skall jag inte ta upp här. Processorn har bara tre register, *ST* (Statusregister), *PC* och *WP* (Registret som pekar ut ackumulatorerna). Det finns även ett register som heter *EIST* som står för *Error Interrupt Status Register*. Detta register ligger bland I/O-adresserna, men finns i processorn.

I EIST kan man avläsa vilken typ av fel som har orskat ett avbrott. Dessa är Aritmetiskt Spill, Odefinierad Instruktion och Oprivilegierat försök att utföra en privilegierad Instruktion. Det faktum att processorn bara har ett register som pekar ut ackumulatorerna gör att man kan ha hur många som helst.

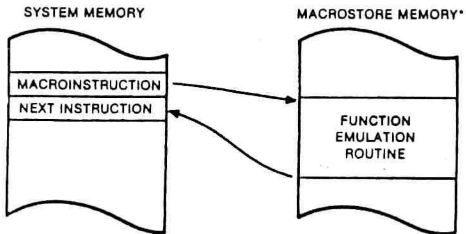


Fig. 4

Men som någon kanske redan har märkt finns det ingen Stackpekare vilket är synd. Med ett eget MACROSTORE kan man få de instruktioner som behövs, en enklare lösning kan vara att man använder ett mjukvaruavbrott.

Avbrott, hårda och mjuka

TMS 99000 har 16 mjukvaruavbrott, sk. XOP:ar. En XOP är en instruktion som hämtar PC och WP från en tabell, sparar undan gamla PC, WP, ST och ett värde i de nya ackumulatorerna. Sedan kan man emulera en egen instruktion som kan vara beroende av det värde man har fått vid anropet.

サイエンス
JAPAN-SCIENCE

Mjukvaruavbrott är inte lika bra som att skapa egna instruktioner då man är begränsad av att man bara får in ett värde. Vid anrop av MACROSTORE får man veta exakt hur den anropande instruktionen såg ut, sedan kan man själv avkoda den som man vill. Det är vanligt att man låter de mest använda subrutinerna ligga som XOP:ar. I Statusregistret finns det en bit som avgör om man skall hoppa till rutinen som står i XOP-tabellen eller om man skall hoppa till MACROSTORE för att ta hand om XOP:én.

TMS 99000 har liksom sin föregångare 16 avbrottsnivåer. Nivå 0 går det inte att skydda sig mot. Alla lägre avbrottsnivåer kan man skydda sig mot genom en avbrottsmask i statusregistret. Avbrottsnivå två genereras både internt och externt, men i EIST kan man avläsa orsaken till avbrottet. I Statusregistret finns en bit som avgör om avbrott för Aritmetiskt Spill skall genereras.

PRIORITY LEVEL	SOURCE AND ASSIGNMENT
LEVEL 0 (HIGHEST PRIORITY)	EXTERNAL RESET SIGNAL
NMI	NON MASKABLE INTERRUPT USER DEFINED
LEVEL 1	EXTERNAL USER DEFINED
LEVEL 2	INTERNAL ILLEGAL INSTRUCTION
	INTERNAL PRIVILEGE VIOLATION
	INTERNAL ARITHMETIC FAULT
	EXTERNAL USER DEFINED
LEVEL 3 – LEVEL 15	EXTERNAL USER DEFINED

Tab. 1

Man kan om man vill istället för att använda MACROSTORE låta en annan processor ta hand om odefinierade instruktioner eller kanske en hel dator. Man kan t.ex. låta en speciell processor ta han om t.ex. signalberäkningar där en TMS 99000 skulle vara för långsam. Eller varför inte ha flyttalsinstruktioner som tas hand av en Flyttalsprocessor. För hittar processorn en odefinierad instruktion så är en av stegen att se efter om det finns en annan processor som vill ta hand om instruktionen.

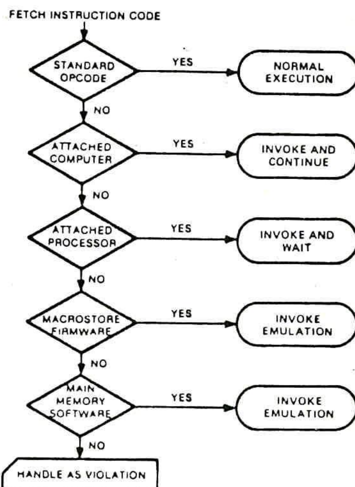


Fig. 5

64 K eller 16 M?

TMS 99000 har liksom Z 8000 endast en adressrymd på 64 Kbyte. Men med en bit i Statusregistret kan man välja en av två sidor vilket gör att man har 128 Kbyte. Man kan göra på samma sätt för MACROSTORE och får då 128 Kbyte minne där. Genom att använda MemoryMapper så kan man få upp till 16Mbyte minne. Men det hade varit trevligare att göra som Motorola har gjort med adresserna på MC 68000 låta det vara mer än 16 adressledningar. Man kan även med hjälp av busskoder lägga upp alla data i en speciell adressarea. Den lösningen finns även på Z 8000 och är lika ful där som på TMS 99000, varför tror dom att man skulle vilja skilja på data och instruktioner? Skriver man ett operativsystem eller liknande så finns det andra sätt att lösa det på.

Om man skall kunna köra **TMS 99000** på maxfart så krävs det att man har väldigt snabba statiska RAM för varje cykel är på 167 ns. **TMS 99000** är ute och läser eller skriver i varje cykel, detta beror på att i den näst sista cykel i varje instruktion görs beräkningar internt, då passar processorn att göra en PreFetch av nästa instruktion. När den instruktionen i sin första cykel avkodar vad det är för instruktion så skrivs resultatet av föregående instruktion ut i minnet. Det är alltså ingen större idé att försöka använd dynamiska RAM.

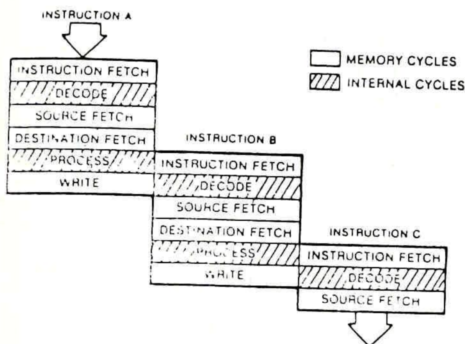


Fig. 6

Programmering av TMS 99000

Som tidigare nämnts så finns det ett antal instruktioner som kräver att användaren har privilegier, se Tab. 2. Instruktionerna CKON, CKOF, LREX och RSET är så kallade externa instruktioner och måste avkodas utanför processorn.

MNEMONIC	DESCRIPTION
CKON	External Clock On
CKOFF	External Clock Off
IDLE	Processor Idle
LIMI	Load Interrupt Mask Immediate
LREX	Load External
RSET	External Reset
LDCR	Load Communications Register Unit
SBO	Set Bit to One
SBZ	Set Bit to Zero
LMF	Load Map File
LDD	Long Distance Destination
LDS	Long Distance Source

Tab. 2

De externa instruktionerna kommer från minidatorn **990/10**, där de även hade en bestämd funktion. De processorer som har kommit efter **990/10** har visserligen haft instruktionerna men de har inte haft någon bestämd funktion, det har varit upp till varje konstruktör att bestämma om man skall avkoda dem.

Till de externa instruktionerna hör även IDLE men till skillnad från de andra instruktionerna så har den en funktion. IDLE lägger processorn att sova, dvs. väntar på att få ett avbrott.

LDCR är instruktionen för att skriva något på en I/O-adress. Som tidigare nämnts så kan man använda instruktionen för en mindre del av I/O-adresserna utan att ha privilegier. Detta gäller även för SBO och SBZ som är skrivinstruktioner. Att läsa från en I/O-adress går alltid bra, även utan privilegier.

LIMI som är instruktionen för att ändra avbrottsmasken är också en instruktion som kräver att användaren har privilegier. Kvar har vi då instruktionerna för att använda 128 Kbyte av minnet. LMF är instruktionen som laddar en MemoryMapper med adresser till sexton block á 4 Kbyte. Det finns två sådana uppsättningar register, det är med LDD och LDS som man kommer åt innehållet i den andra uppsättningen register.

Genom att utföra LDD och/eller LDS så kommer "*Destination*" resp. "*Source*" att hämtas från det alternativa minnet. LMF, LDD och LDS räknas till standard-instruktionerna men finns bara med på de processorer som har MACROSTORE. De finns alltså inte på TMS 99105, då den saknar MACROSTORE.

I MACROSTORE-program har man även tillgång till en del special instruktioner. Av dem är väl EVAD, *Evaluate Address*, den som är trevligast. Den räknar ut två adresser ur innehållet i ett register. I registret skall vara den del av instruktionen som innehåller adresseringsmoderna. Använder man TI:s adresseringsmoder i sina egna instruktioner, så krävs det väldigt lite överflödig kod vid emulering i MACROSTORE.

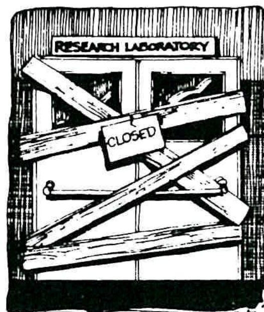
TMS 9900 var en av de första μ-processorerna som hade multiplikation och division i hårdvaran. Instruktionerna, MPY och DIV, var endast tänkta för positiva heltal. För tvåkomplementstal så krävdes det dock ett litet program, på ca: 7-8 rader kod. Men TMS 99000 har även fått instruktioner för tvåkomplementstal (MPYS och DIVS). TMS 99000 har även fått instruktioner för addition och subtraktion av 32-bitars heltal.

TMS 99000 har även fått instruktioner för att hantera en bit i ett minnesord. Detta saknades, tyvärr, i TMS 9900. Det är konstigt hur ofta man behöver bitmanipulering när man inte har det.

Kan man tro på en Benchmark?

När jag har läst Benchmark-rapporter från TI om TMS 9995 och TMS 99000 så har jag förundrat mig över hur mycket snabbare TI:s processorer är än konkurrenternas. Det beror nog på att man väljer en Benchmark som passar processorn man har. I båda fallen har det varit Benchmark-rapporter från Intel. Jag tror inte att TMS 99000 är nästan dubbelt så snabb som en 10 MHz MC 68000. Med den i EDN (April 1981) publicerade Benchmarkrapporten så skulle TMS 99000 vara 25% snabbare än MC 68000, enligt TI. ∎

Mats O Jansson



Litteratur

[1] TEXAS INSTRUMENTS
"TMS 99000 Family, 16-Bit Microprocesssor" Preliminary Product Specification, November 1981

[2] TEXAS INSTRUMENTS
"Microsystems Designers Handbook" 2nd Ed. 1981, ISBN 0-904047-33-4

[3] GRAPPEL R D, HEMENWAY J E
"A tale of four µPs: Benchmarks quantify performance." EDN April 1, 1981. p. 179-265

Motorola 6805

Hårdvara

Detta är en c:a två år gammal familj av enchips-processorer. I denna familj ingår både ROM och EPROM-versioner, där den äldsta (och därmed mest lättillgängliga) EPROM-versionen heter **68705P3** och har 112 bytes RAM och 1.8 Kbyte EPROM. En av dess trevligare egenskaper är att den själv programmerar in sitt EPROM-innehåll (från exempelvis ett **2516**), dvs det behövs väldigt lite hårdvara för att programmera den.

Mjukvara

6805 har en 8-bitars accumulator och ett 8-bitars indexregister, en stackpekare (i **68705P3**:s fall 6-bitars !), en programräknare (i **68705P3**:s fall 11-bitars !!) samt ett statusregister. Instruktionerna består av:

- Bittest-instruktioner som hoppar (—128 - +127 bytes) om en viss bit i adressområdet 0-255 är ett eller noll-ställd.
- Bitclear/Bitset-instruktioner som kan manipulera motsvarande delar av minnet.
- Relativa hopp (—128 - +127 bytes) beroende på ett av 16 villkor.
- Enoperands-instruktioner (t ex **NEG**era) som kan adressera accumulatorn, indexregistret, minne direkt (men bara på adress 0-255) och indexerat (indexregistret med eller utan offset).
- Tvåoperands-instruktioner med en implicit operand (vanligen accumulatorn eller indexregistret) (t ex **ADD**era) som kan adressera hela minnet direkt och indexerat (indexregistret med 8- eller 16-bitars offset eller helt utan offset).
- Vissa specialoperationer utan operand (t ex **NOP**).

Tyvärr ingår inte Push/Pop av register, men det är en av de få saker man verkligen saknar. (Det går nog att komma runt i och för sig eftersom instruktionen SWI (*Software Interrupt*) som bl a lägger upp registerna på stacken finns.) Totalt sett tycker jag att **6805** är en mycket trevlig processor för små (och kanske lite större) controllerapplikationer. ■

Dan Norstedt

SOFTNET - lägesrapport

Kommer du ihåg amatördatanätet SOFTNET som vi beskrev i nummer 1 detta år av STACKPOINTER? Ett projekt vid linköpings universitet med målet att konstruera hård- och programvara för ett dataöverföringsnät med radioöverföring.

Det som då var långt drivna planer har nu resulterat i att både mikrodatordelen och sändare/mottagare-delen är så gott som färdigkonstruerade. Innan byggbeskrivning, byggsatser eller färdiga noder kan börja levereras återstår att hitta firmor som kan leverera erforderliga komponenter till hyggligt pris.

Ett projekt på icke-kommersiell grund

SOFTNET är ett amatörorienterat projekt. Inga företag skall kunna ta kontrollen över det och förhindra insyn eller skaffa sig ensamrätt till SOFTNET-systemet. Det skall dock vara fritt fram för olika företag att tillverka och sälja noder och tillhörande hårdvara, men kärnan i SOFTNET-gruppen kommer att se till att information och erfarenheter är fritt tillgängliga för alla som använder systemet.

SOFTNET user group

En SOFTNET-förening håller på att bildas i Linköping. Den kommer att fungera som en förbindelselänk mellan SOFTNET-konstruktörer, användare och alla som är inresserade av projektet. Då de första noderna kommit igång kommer mycket av arbetet med SOFTNET att gälla nodernas programvara. Här kommer SOFTNET-föreningen fungera som ett forum för idéutbyte kring just programmeringsfrågor.

Personligen tror jag att en av de första viktiga SOFTNET-tillämpningarna vore ett konferenssystem av typ KOM, där medlemmarna deltar via sina SOFTNET-noder. En någorlunda centralt placerad nod med dator och massminne kunde då innehålla KOM-databasen. Ett lämpligt STACKEN-projekt!

Idiotsäker radiodel

Radioamatörer har sedan många år använt grejor från datoramatör-området i sin verksamhet. SOFTNET kommer att innebära det motsatta, att datoramatörer utnyttjar radioamatör-grejor. Detta kan vålla problem eftersom man ju mycket väl kan vara duktig datoramatör utan att veta ett dugg om vågutbredning, modulation eller filter. Hur skall en datoramatör klara att sköta en SOFTNET-nod, som innehåller så mycket analog-prylar?

Lösningen är att SOFTNET-radion konstrueras så att den inte behöver nästan någon trimning alls för att fungera tillfredställande. Till de kritiska delarna i radion har man därför i möjligaste mån använt fasta fabrikstrimmade komponenter, t.ex. kristallfilter. Detta medför också att hem-byggaren inte behöver ha tillgång till en massa dyra instrument – spektrum-analysatorer etc. – för att få noden att funka. Det är dessa komponenter som har varit svårast att få tag i till vettiga priser och med rimliga leveranstider.

Televerkets krav

Som jag skrev i förra artikeln om

SOFTNET kräver televerket amatörradiotillstånd för de som skall använda SOFTNET-noder. Principen är egentligen att alla som utnyttjar *radiosändare* måste vara registrerade hos televerket. Praktiskt löses detta genom att man tar T-certifikat. Detta certifikat ställer inga stora krav på radiokunskaper och inga alls på morsetelegrafering.

Men för datoramatörer som utnyttjar en helt fixkopplad, opåverkbar, typgodkänd standardsändare borde det inte vara så viktigt att veta att QRQ 17 betyder "sänk er telegraferingshastighet till 17 ord per minut", eller att en superheterodynmottagare innehåller en variabel lokaloscillator! SOFTNET-gruppen har påtalat detta för televerket, men det står fast vid sitt T-krav.

Men Dator-amatör, låt dig inte tryckas ner! Det är ingen match för dig att ta T-cert. Du har ju lärt dig svårare koder än Q-koden, -eller hur?

Och det är glädjande att både televerket och förbundet Sveriges Sändar-Amatörer ställt sig så positiva till SOFTNET att projektet blivit möjligt.

En vanlig fråga jag får då jag försöker entusiasmera datorister för SOFTNET, är om televerket verkligen tillåter att en massa information överförs kors och tvärs mellan SOFTNET-noderna på ett okontrollerbart sätt. SOFTNET-noder *vidarebefordrar* ju meddelanden på ett sätt som är förbjudet för vanliga radioamatörer. Men televerket har givit SOFTNET dispens på denna punkt. Ett annat krav på radioamatörer är att de minst var tionde minut under en förbindelse måste identifiera sig genom att sända sin anropssignal. Detta är inget problem för SOFTNET-noder. Det ingår i deras funktionssätt att identifikation skickas med varje datapaket, som ju ofta sänds flera tusen gånger per sekund. Det gäller bara för televerket att hinna med och uppfatta detta. SOFTNET-gruppen kommer att konstruera utrustning som televerket kan använda för att övervaka trafiken med.

SOFTNET-noden

I sitt nuvarande utförande består en SOFTNET-nod av två huvuddelar

- Sändare/mottagare
  - frekvens 434.000 MHz
  - sändareffekt 7W (programstyrd)
  - mottagare med programstyrd känslighetströskel
- Styrdator med 2 st 6809
  - en CPU styr sändare/mottagare
  - en CPU sköter kommunikations-strategier och anpassning till yttre datorer/terminaler
  - programmeras i speciell multiprocess-FORTH

Dataöverföringshastigheten är 100Kbit/s maximalt, men den effektiva hastigheten är lägre beroende på hur mycket nod-trafik som pågår, väderleksförhållanden m.m.

Intresse från företag och organisationer

Utvecklingen av SOFTNET följs med nyfikenhet från många olika håll. ERICSSON (Tre Korvar), SRA, SATT och FOA har visat stort intresse. FOA har redan på eget initiativ konstruerat och byggt en nod som skall kunna delta i ett SOFT-nät. STU är också med och finansierar delar av projektet. ∎

Dag Rende

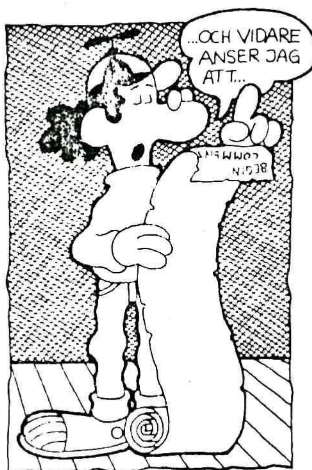
Gå med i SOFTNET users group, så får du löpande information om projektet. Skriv namn och address på en lapp och skicka till: Inst För Systemteknik, Att: Jens Zander ISY, Linköpings Tekniska Högskola, 581 83 Linköping.

Insändare

Herr Redactörn

Hur kan en san förening som STACKEN innehålla sådana medlemar, som med berått mod bara MÖRDAR fredliga indrivare av föreningens skulder. Hade dom egna skulder, eller ville dom bara inte ha en sådan personlighet som Alexander i föreningen. Men framfor allt ska vi fortsätta vår kamp för mögelsvamparnas likaberättigande. Som vårat första mål har vi satt upp HEDERSMEDLEMSKAP ÅT ALEXANDER.

Föreningen för Mögelsvamparnas
Likaberättigande



The Last Bug

"But you're out of your mind"
they said with a shrug.
"The customer's happy-
what's one little bug"

But he was determined.
The others went home.
He spread out the program,
deserted, alone.

The cleaning men came,
the whole room was cluttered
with memory dumps
"I'm close", he muttered.

He finally shouted:
"Simple deduction!
I've got it, it's right,
just change one instruction."

It still wasn't perfect
as year followed year,
and strangers would comment:
"Is that guy still there?"

He died on the console
of hunger and thirst.
Next day he was buried
six feet, nine edge fist.

And the last bug in sight,
an ant passin by
saluted his tombstone
and wispered: "Nice try!"

Author Unknown

Du ser det gamla paret här!
Fru Z och herr Q det är.
De är så rysligt gamla så,
man snart glömt bort dem båda två.
Förr skrev man ofta K med Q,
och S med Z skrev man ju.
Men nu de finnas bara kvar
i några namn från gamla da'r.
Jag tror, jag såg dem allra sist
i namnet Zetter-qvist.