PROGRAMVARAN I PERSONDATORN ABC80

Examensarbete givet vid

Inst för tillämpad elektronik
Kungl Tekniska Högskolan

Utfört av

Johan Finnved

Godkänt den: 29 jan 79

O. Mattsson

Programvaran i persondatorn ABC80

****

Examensarbete av Johan Finnved

EXAMENSARBETE FÖR JOHAN FINNVED (0773-0856)

PROGRAMVARAN I MIKRODATORSYSTEMET ABC 80

Arbetet skall ge en beskrivning av olika BASIC-nivåer. En BASIC lämplig för persondatorbruk skall tas fram. Den skall bestå av följande delar:

1. Initiering
2. Huvudprogram med kommandoavkodare
3. Kommandorutiner. exempelvis LIST, RUN
4. Internkodskompilator
5. Uttrycksberäknare
6. Exekveringsrutiner. exempelvis PRINT, GOTO
7. In- och utmatningsrutiner inklusive diskoperativsystem

Arbetet utföres vid Dataindustrier AB, Täby.

Arbetet startat 1978-02-15

Handledare: Gunnar Markesjö

Inledning

Persondatorn ABC80 programmeras liksom alla konkurrenter på marknaden i BASIC. BASIC är inte det bästa språket för alla användningar, men det har den klara fördelen att det är lätt att lära sig. Dessutom har BASIC den fördelen att det är lätt och ganska naturligt att köra BASIC interaktivt så man kan lätt ändra i program och testa direkt utan att gå via kompilering eller assemblering.

Eftersom BASIC är det språk som de flesta lär sig först är det mer eller mindre självklart att persondatorer skall programmeras i BASIC. I och med persondatorernas inträde kommer väl orsak och verkan snart att byta plats.

Låt oss därför först studera några sätt att göra BASIC-system!

För att kunna ordna in ABC80's BASIC i rätt fack kan vi generalisera lite bland de BASIC-interpretatorer som finns.

Indelning med avseende på datatyper.

1. Den ursprungliga definitionen på BASIC omfattade endast flyttal som datatyp att räkna med. De första och enklaste BASIC systemen klarade också bara detta. Eftersom en stor del av databehandling omfattar hantering av texter måste dessa BASIC-system betraktas som rena leksaker.
   Exempel: Processor Technology's 5k BASIC.

2. De flesta BASIC-interpretatorer kan förutom flyttal även hantera textsträngar - ett måste för att kunna få ordentliga möjligheter att klara kommandoavkodning och överhuvud taget göra lite mer avancerade program.
   Exempel: Imsai's 8k BASIC.

3. Ett antal BASIC-interpretatorer kan förutom flyttal och strängar dessutom hantera heltal. Alla operationer med heltal kan naturligtvis också ske med flyttal som ju också innefattar heltalen som delmängd. Det tar emellertid tio till hundra gånger så lång tid, beroende på om flyttalsaritmetik är realiserad med hård- eller mjukvara, så för att få program att gå fortare använder man gärna heltal på alla ställen där man inte verkligen behöver flyttalen.
   Exempel på BASIC-interpretatorer med alla tre datatyperna är Digitals BASIC plus (Elviras BASIC) och även ABC80's.

   Både i ABC80's och i Elvira BASIC finns en extra finess nämligen ASCII-aritmetik. Detta innebär att man kan räkna med tal i textsträngar med mycket större precision än både flyttalen och heltalen har. ASCII-aritmetik är nödvändig till exempel vid administrativa beräkningar där man annars snart får fel vid öresberäkning.

Indelning med avseende på grad av kompilering.

1. Det finns BASIC-interpretatorer som inte gör någonting alls åt inmatade rader. Med andra ord lagras ett ord som t.ex. GOTO med alla fyra bokstäverna. Med denna metod måste ordet kännas igen varje gång man kommer till det vid exekveringen, och man får naturligtvis en ganska effektiv broms på exekveringen. Dessa BASIC-interpretatorer måste betraktas som skräckexempel.
   Exempel: Imsai's 8k BASIC.

2. Något mer avancerade BASIC-interpretatorer byter ut de kända nyckelorden mot interna opkoder men gör för övrigt ingenting åt det inmatade programmet. Här slipper man jämföra texter vid exekveringen, men alla andra beslut belastar exekveringen, dels att kontrollera att raderna är syntaktiskt riktiga och dels i vilken ordning de

matematiska operationerna skall ske i uttryck. Detta bromsar också exekveringen en hel del.
Det finns en egenskap som omedelbart avslöjar en BASIC av typ 1 eller 2. De accepterar vad som helst som satser vid inmatningen av program och protesterar först vid provkörning även om raden lyder '10 SOPOR'.

3. Nästa steg är den stora skara av BASIC:ar som förbehandlar satserna enligt 2 men som dessutom kontrollerar att raderna är syntaktiskt riktiga redan vid inmatningen. Man får alltså omedelbar diagnos på alla fel som går att hitta på en isolerad rad. Även hos dessa BASIC:ar sker besluten om i viken ording ett uttryck skall beräknas under exekveringen. Vanligen har man heller inte adresser till refererade rader och variabler utan bara radnummer och variabelnamn i den lagrade koden. Sökning efter referenser vid GOTO och vid användning av variabler belastar fortfarande exekveringen. Detta är troligen den allra största gruppen av BASIC-interpretatorer.
Exempel: Interdata's BASIC II, Data General's Multi User BASIC för Nova.

4. Nästa grad av kompilering innebär att uttryck har koverterats till omvänd polsk notation (RPN används även på HP's fickräknare) dvs opkoderna för uttryck kommer i den ordning de skall behandlas. Exempel: 4+5*6 konverteras till Push4, Push 5 , Push 6 , Mult, Add. Dessutom införs adresser till referenser till radnummer och variabler. Exempel: ABC80.

5. Ännu längre driven kompilering intill olistbarhet förekommer. Här preprocessas raderna till en följd av opkoder där man inte längre kan härleda hur det BASIC-program såg ut som genererade koden. Man har alltså två kopior av BASIC-programmet: Dels den ursprungliga texten helt oprocessad, allså som pkt 1, och dels den kompilerade koden. I interaktiva system av denna typ har man texten på en temporärfil på en snabb disk, så att man inte behöver belasta arbetsminnet med mer än en kopia av programmet. Den konventionella metoden att åstadkomma en helt kompilerad kod är att editera hela programmet med en texteditor till en diskfil, som man sedan kompilerar till en objektfil som kan köras efter ev länkning. Detta ger möjligheter för kompilatorn att på en gång reda ut relationer mellan satser eftersom den läser hela programmet i rätt ordning. I interaktiva system är detta inte möjligt eftersom interpretatorn inte vet i vilket sammanhang en rad kommer att finnas då programmet är färdigt. Det tar oftast längre tid att utveckla program i system som inte är interaktiva eftersom man alltid måste gå tillbaka till editorn och sedan göra alla stegen om igen till ett körbart program vid rättning av smärre fel.
Exempel på interaktivt system: Digital's Basic plus (Elvira).
Exempel på ej interaktivt system: Basicen i UCSD's pascal system.

Indelning med avseende på strängfilosofi.

I BASIC-system som klarar textsträngar finns en ganska markerad skiljelinje mellan två filosofier. En som man ibland kallar HP:s filosofi, och en som man kallar Digital's filosofi.

1. I HP's filosofi betraktar man strängar som vektorer där elementen är tecken. Man kan därmed tilldela hela eller delar av en strängvariabel nya värden. Eftersom indicering redan är upptagen kan man inte

(åtminstone inte på ett naturligt sätt) ha flera oberoende strängar som man når med index.

2. I Digital's filosofi betraktar man alltid strängar som enheter, och man har speciella funktioner för att nå substrängar. Index används inte för att beteckna substrängar utan används i stället för att välja ett element i en vektor eller matris av strängar.

Valet mellan de två filosofierna är tämligen godtyckligt och är egentligen en smaksak. Jag har gjort valet att använda Digital's filosofi eftersom jag tycker att den passar bättre in i BASIC. Hanteringen av index blir också mera konsekvent i detta fall. Elvira har väl också indoktrinerat en del. Man måste dock understryka att man knappast kan påstå att någon filosofi är entydigt bättre än den andra. Att Digital's filosofi kommit till användning avspeglar alltså min smak (som jag därmed kanske pådyvlar några andra).

1. Initiering.

Initieringen av ABC80 sker vid spänningstillslag, då resetknappen på baksidan trycks in eller då något program som körs hoppar till adress noll. Detta sker endast då katasrofala fel hittas som omöjliggjör forsatt körning. Först initieras periferikretsarna på ABC80 kortet och ABC80-bussen. ABC80-bussen ges resetsignal RST ut (görs även i hårdvara vid spännigstillslag eller tryck på knappen, men inte då hopp görs till noll). Z80 CPU:n sätts i interrupt mode två, vektoriserat interrupt. Båda PIO-kretsens halvor sätts i bit mod.

PIO del A sätts i bit mod för att tangentbordsstroben skall kunna ge interrupt och även läsas statiskt (används av automat-repeat se kap 7). Man hade inte kunnat använda parallell input mod eftersom data då latchas och stroben hade inte kunnat avkännas statiskt. Dessutom blev på det här sättet strobe-ingången ledig och den används som snabb klocka (128 mikrosekunder) vid t.ex. kommunikation.
Del B måste vara i bitmod eftersom det finns både in- och utgångar från den PIO-halvan.

Ram-minnets storlek kollas sedan uppifrån och nedåt genom att en byte i varje bank (256 bytes) läses in, komplementeras, skrivs tillbaka och kontrolläses. Om en byte låter sig komplementeras anser man att det finns minne i den banken. Efter varje provskrivning återställs byten till orginalvärdet för att testen inte skall förstöra lagrad information i minnet. Även om man har mycket minne stoppas sökningen i adress 8000 Hex så att inte bildminnet tas i anspråk för programlagring.

Bildminnet städas dvs fylls med blanktecken.

Nu kontrolleras om man har anslutit floppyoption. Kriteriet på att floppyoption finns är att den börjar med instruktionen JMP. Om det inte finns någon floppyoption ordnar pull-up motstånden på databussen att man läser 255 decimalt i stället. Om optionen finns görs ett anrop av dess initieringsrutin. Detta anrop innebär att ABC80:s diskoperativsystem tar sex minnesbanker (6*256 bytes) från BASIC till filbuffertar. Sedan initieras operativsystemets interna tabeller och ett försök görs att öppna filen 'BASICERR.SYS' på alla

1. Initiering.

minifloppyenheter. Om försöket lyckas hämtar BASIC i fortsättningen sina felmeddelanden i klartext från denna fil.

Efter det eventuella besöket i floppyoptionen simuleras ett SCR-kommando och sedan skrivs 'ABC80' ut på skärmen och ABC80 står i det vanliga inmatningsläget.

2. Huvudprogram med kommandoavkodare.

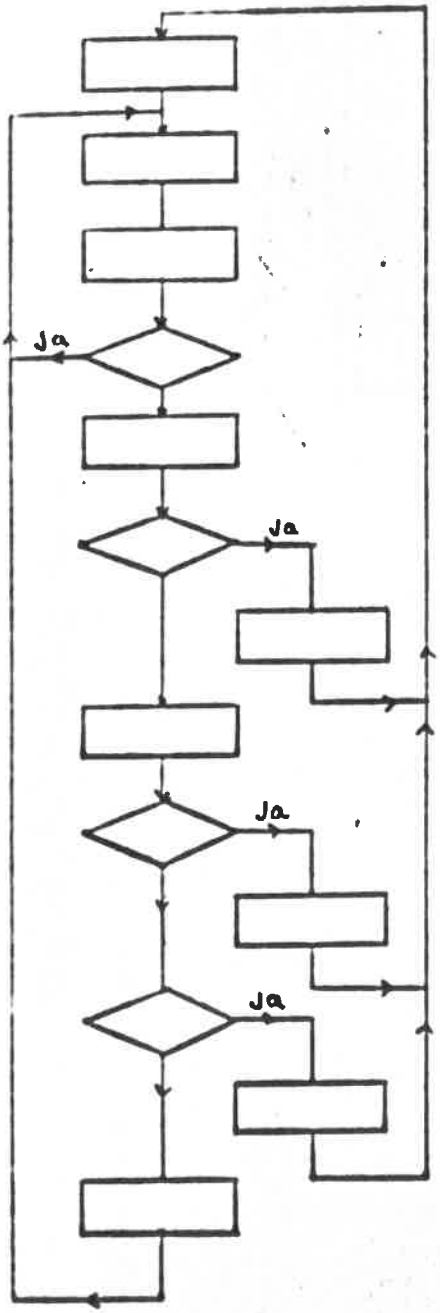
Huvudprogrammet ligger man i då inte ett program exekveras. Här skrivs 'ABC80' ut som prompt för varje gång kommando väntas (utom då en programrad just matats in), och en rad läses från tangentbordet.

I första hand antas att den inmatade raden är ett kommando, så början på raden jämförs med kommandotabellen. Om raden var ett kommando anropas rutinen för kommandot i fråga utan att raden analyseras vidare.
Om inte raden stämmer med något kommando kompileras den av satskompilatorn, se kap 4.
Om fel upptäcks vid kompileringen ges felmeddelande i klartext om floppyoption och errorfil finns, annars i form av felnummer.

Om raden var korrekt och hade satsnummer sorteras den in i det lagrade programmet och en ny rad efterfrågas utan någon prompt-text.
Om raden var korrekt men inte hade något radnummer, så görs först fixup (Variabelnamn mm fylls i. Se vidare under rubriken RUN i kap 4) på den nyligen kompilerade raden som sedan exekveras direkt.

Se även flödesplan på nästa sida.

Flödesshema huvudprogram



Skriv ut 'ABC80'.

Gör radframmatning.

Läs in en rad från tangentbordet.

Helt tom rad ?

Leta i kommandotabell.

Hittade ett kommando ?

Utför kommandot (anropa tillhörande rutin).

Uteslutningsmetoden säger att det måste vara en sats - Kompilera den.

Fel hittat vid kompileringen ?

Skriv ut meddelande (nummer eller klartext).

Sats med radnummer ?

Utför den nyligen kompilerade satsen.

Sortera in satsen i det lagrade programmet.

3. Kommandorutiner LIST & RUN

Då huvudprogrammet hittar ett kommando anropas tillhörande rutin utan att raden analyseras vidare. Av detta skäl kan man inte tillåta uttryck i kommandon eftersom de ju först skall kompileras och fixas. Å andra sidan behöver textsträngar i filnamn inte avgränsas av citationstecken eftersom inte någon kompilator behöver få veta att texten skall tolkas som textkonstant och inte som variablenamn e dyl.

Alla kommandorutiner anropas med Z80:s HL register pekande på första icke mellanslag efter kommandoordet och sedan är det kommandorutinens uppgift att, om det behövs, ta hand om parametrar.

Exempel på kommandorutiner:

LIST

Eftersom ett registerpar pekar på första parametern kontrolleras lätt om första tecknet är en bokstav eller ett citationstecken. I så fall antas att ett filnamn avses och en listfil öppnas ('PR:' betraktas också som fil). Även om citationstecken inte behövs hos filnamn i kommandon finns det ju ingen anledning att förbjuda dem. Om inget filnamn hittades noteras att listning skall ske till bildskärmen och en radräknare initieras till 22. Det är denna räknare som används för att få listningen att vänta på en tangenttryckning efter en sida.

Vidare, i fallet bildskärmen, tas radnummer om hand som gränser för listningen. Om inget ges eller om listan skickas till fil sätts gränserna till 0 respektive oändligheten (=65535). Första raden letas upp och sedan har vi nått LIST-loopen.

I loopen kontrolleras först om vi har kommit till programmets slut eller operatören har tryckt CTRL-C. I så fall är det dags att sluta lista och stänga ev listfil. Om programmet inte är slut finns det alltså en rad vars nummer jämförs med den givna övre gränsen, och om radnumret är större så avbryts också listningen.

Om alla dessa tester passeras anropas rutinen för baklänges kompilering av raden. Detta innebär bl.a. att alla uttryck omvandlas tillbaka från omvänd polsk notation (RPN) till vanlig infix notation som vi är vana att läsa. Efter denna baklängeskonvertering matas raden ut till konsol eller till fil (printer).

Sist i LIST-loopen ligger nedräkning av radräknaren (endast konsol) och då den når noll görs enteckensinmatning och återställning av radräknaren till ett för att ge användaren en chans att hinna läsa listan.

RUN

RUN-kommandot är naturligtvis det viktigaste kommandot. Det är ju via detta som man kommer åt att exekvera programmet man har redigerat.

Först i RUN-rutinen kontrolleras om det står något efter 'RUN'. I så fall anropas LOAD-rutinen så att angivet program laddas. Nästa anrop görs till CLEAR-rutinen så att hela variabelarean initieras och

alla filer stängs. Sedan anropas fixup som fyller i alla adresser till variabler och satsnummerreferenser mm.

Fixuprutinen läser egenom programmet en gång och stannar på vissa opkoder. Radnummer (vid exempelvis GOTO) leder till at raden letas upp och adressen till den fylls efter numret. Variabler letas upp i variabeltabellen, och om den inte hittas så läggs den till på slutet. Adressen till variabeln lagras efter namnet i programmet. Då ett anrop av användar-funktion påträffas letas funktionens definition upp, och argumenten kontrolleras med avseende på antal och typer. Konvertering genereras mellan heltal och flyttal automatiskt. Adressen till definitionen läggs efter namnet.

Då en FOR-sats påträffas läggs en nod på FOR-stacken. Noden innehåller pekare till första satsen efter FOR och loopvariabelns namn. Då en NEXT påträffas tas den översta posten på FOR-stacken bort och kontrolleras. Stacken får naturligtvis inte vara tom, och variablerna måsta stämma. Nu finns pekare till båda ändar på loopen tillgängliga, och FOR- och NEXT-satserna knyts samman genom att de förses med pekare till varandra.

Då hela programmet avverkats kontrolleras att FOR-stacken är tom, annars finns det ju en FOR utan tillhörande NEXT.

FOR-stacken finns bara under fixupfasen. Under exekveringen ligger variablerna för stegvärde och slutvärde på fixa ställen i minnet utanför stacken. Detta gör att man kan hoppa ur FOR-loopar utan att man behöver bekymra sig över att tömma FOR-stacken.

Alla fel som rör programmets struktur hittas nu. (Fel av typen radnummer finns inte men refereras vid GOTO, eller att fel finns i FOR-loopar.)

Härnäst anropas RESTORE-rutinen så att nästa READ-sats läser från programmetas första DATA-sats. Sista initieringsrutinen som anropas är en rutin som initierar slumptalet till noll så att man alltid får samma följd av slumptal (åtminstone till dess en RANDOMIZE-sats utförs).

Sist PUSH:as en pekare till felhanteraren för 'Return without Gosub' så att en otillåten RETURN ger felutskrift. Nu laddas interpreteringspekaren (Z80:s DE) med början på programmet och körningen kan börja. Vi kommer in i RUN-loopen som genomlöps en gång per rad.

Först i RUN-loopen kontrolleras om programmet är slut. I så fall anropas rutinen för END så att alla filer stängs och variabler städas.

Vi antar att programmet inte är slut. I så fall sparas interpreteringspekaren så att felhanteraren kan skriva ut på vilken rad fel uppstår. Sedan kontrolleras om TRACE pågår och i så fall skrivs radnumret som just passeras ut.

Nu utförs alla satser som står på raden i en inre loop, och mellan varje sats kontrolleras om operatören har tryckt på CTRL-C.

4. Kompilatorn

BASIC interpretatorn är konstruerad med tanke i första hand på snabbhet. Det innebär att så många beslut som möjligt skall vara gjorda innan programmet körs. För att åstadkomma detta sker en kompilering av raderna då de matas in och alla beslut om huruvida raden är korrekt och i vilken ordning de ingående operationerna skall utföras sker här. Eftersom programmet skall gå att lista kan dock inte kompileringen bli fullständig utan viss information som är onödig vid exekveringen måste kvarstå. Raderna i ABC80-basic får ju matas in i godtycklig ordning så den mera övergripande strukturen hos programmet kan man inte göra något åt här.

För att inte belasta exekveringen med att söka efter variabler, användarfunktioner eller rader vid t.ex. GOTO och FOR-NEXT sker det ett pass, kallat fixup, då man skriver RUN men innan första raden exekveras då alla adresser fylls i. Eftersom alla variabler och satser refereras direkt med adresser kan programmet sedan exekveras snabbt. Hur fixup går till beskrivs i kap 3 avsnittet om RUN.

För att bibehålla listbarhet blir det ett visst slöseri med minnesutrymme, både radnummer respektive variabelnamn och adress lagras.

Kompilering av uttryck

Den mest systematiska delen av kompileringen är den av uttryck. Här översätts uttryck i den form vi är vana att se dem (infix notation) till postfix notation (även kallad ovänd polsk notation eller Lukaciewitz notation), av samma typ som även används av vissa fickkalkylatorer.

Postfix notation är mycket effektiv vid exekveringen eftersom varje element svarar mot en operation precis i den ordning de uppträder. Alla de ovan nämnda operationerna sker mot en stack, d.v.s. en minnesarea som används som sist in först ut buffert.

Följande exempel visar principen: Uttrycket 5*(4+3) översätts till 543+* där siffrorna motsvarar operationen lägg konstant överst på stacken. De matematiska symbolerna opererar på de två översta elementen på stacken och resultatet ersätter dessa två argument.

För att kunna använda variabler behövs instruktioner som flyttar data mellan stacken och variabelminnet. För indexerade variabler måste man dessutom kunna räkna ut verkliga adressen. För att reducera antalet nödvändiga operatorer har jag delat upp funktionen i två operationer, en som lägger adressen till variabeln på stacken och en som utför PUSH eller POP. PUSH får då följande funktion: Elementet överst på stacken betraktas som adress och byts mot värdet som adressen pekar på. POP får funktionen: Värdet som ligger överst på stacken läggs där adressen som ligger näst överst på stacken pekar.
Följande exempel visar översättningen av ett uttryck med vektorelement till motsvarande RPN-uttryck.
A(4%)=A(3%)+B översätts till Const 4, Vector A, Const 3, Vector A, PUSH, Scalar B, PUSH, Add, POP. Märk att adressen till A(4) ligger underst i stacken ända fram till den sista POP-instruktionen. Märk

även att med denna funktion på POP kan kompilatorn lägga ut hela uttrycket för vänsterledet före uttrycket för högerledet, dvs i den ordning det förekommer på raden. Kompilatorn slipper alltså spara uttrycket tills POP-instruktionen skall genereras.

Kompilatorn är en recursive descent kompilator där varje element i språket motsvaras av en procedur som tar hand om just det elementet. För att visa metoden förenklar vi uttrycken kraftigt. Vi antar att att vi bara har de fyra räknesätten +-*/ opererande på konstanter. Syntaxen för uttryck kan då illustreras av syntaxdiagrammen i figur 2. Som synes är definitionen rekursiv: uttryck ingår ju i definitionen för faktor.

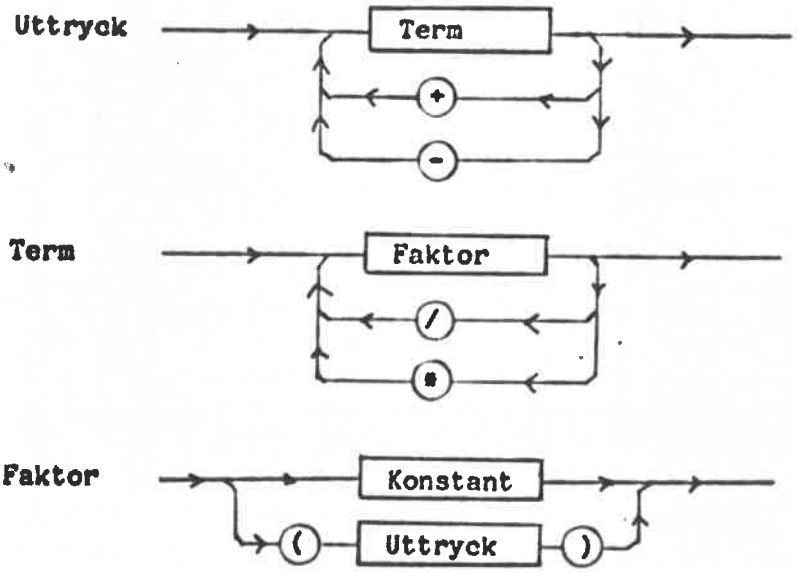


Fig 2 Syntaxdiagram förenklade uttryck.

En kompilator för detta kan formuleras sålunda i exempelvis Simula. T antas innehålla aktuellt tecken, proceduren nextch hämtar nästa tecken.

```
PROCEDURE Expression ;
BEGIN Term ;
  WHILE (t='+') OR (t='-') DO
    BEGIN CHAR t1 ; t1:=t ; t:=nextch ;
      Term ; Gen(IF t1='+' THEN 'ADD' ELSE 'SUB' )
    END while
END Expresion ;


PROCEDURE Term ;
BEGIN Factor ;
  WHILE (t='*') OR (T='/') DO
    BEGIN CHAR t1 ; t1:=t ; t:=nextch ;
      Factor ; Gen(IF t1='*' THEN 'MUL' ELSE 'DIV')
    END while
END Term ;
```

```
PROCEDURE Factor ;
BEGIN IF t='(' THEN
    BEGIN t:=nextch ; expression ;
      IF t<>')' THEN  Error ELSE t:=nextch ;
    END Parentes med rekursivt anrop av expr.
  ELSE
    BEGIN <Tag hand om talet.> ; Gen('CONST',value)
    END Konstant ;
END Factor ;
```

Syntaxdiagrammet översätts alltså tämligen direkt till procedurer. Märk att man helt slipper hålla reda på operatorernas prioritet och man slipper att räkna parenteser, detta sker helt automatiskt tack vare programstrukturen.

Som synes är procedurerna Expression och Term mycket lika och att båda finns är bara ett sätt att uttrycka prioritetsordningen mellan operatorerna. I ABC80-BASIC finns nio prioriteter för operatorerna (se manualer). Det vore oekonomiskt att ha nio kopior av i princip samma procedur. I stället används en och samma procedur, och den anropas med argument som anger vilken vilka tecken som man skall jämföra med och vilken procedur som skall anropas i nästa steg.

Hittills har vi inte betraktat det faktum att det i ABC80-BASIC finns tre datatyper, nämligen flyttal, heltal och textsträngar. För att klara att generera rätt version av operatorer, kontrollera att en otillåten blandning inte förekommer och generera nödvändiga konverteringar mellan hel- och flyttal tillämpas generellt att varje procedur av ovanstående typ rapporterar vilken typ resultatet får. Ansvaret att kontrollera typer och skjuta in konverteringar faller då hela tiden på den anropande proceduren. I det nämnda förenklade exemplet konstaterar proceduren Factor av vilken typ konstanterna är och rapporterar detta till Term. Proceduren term å sin sida sparar den typ första anropet till Factor gav och kollar vid nästa anrop att den får en typ som passar till den första (antingen exakt samma eller åtminstone kompatibla). Om typerna inte var samma men kompatibla så genereras erforderlig konvertering.

I ABC80 gäller följande regler:

Om en operand är sträng så måste den andra också vara sträng och den enda tillåtna operationen är addition (konkatenering).

Om operationen är logisk konverteras båda leden till heltal och operationen anses ge heltalsresultat.

Om operationen är aritmetisk utförs den som heltalsoperation endast om båda leden är heltal. Om endera operaden är flyttal konverteras alltså den andra operanden till flyttal och operationen utförs som flyttalsoperation.

En komplikation tillstöter vid fallet A%+B. Här kompileras vänstra operanden och ger typen heltal. Sedan kompileras högra operanden och ger resultat flyttal. Det är nu egentligen för sent att mata ut konvertering heltal-flyttal. Resultatet skulle bli PUSH A%, PUSH B, Cvt%toF (Bortse från att PUSH variable består av två steg!). Konverteringen skulle operera på A% som är begravt i stacken av B. Då

konverteringen innebär en ändring av storleken av datat den opererar på skulle B behöva flyttas och hela operationen blir tydligen omständig vid exekveringen.

Bättre är att skjuta in konverteringen direkt efter koden för vänsterledet. Detta problem löses genom att alla procedurer sparar pekare till var koden för vänstra operanden slutar innan anrop görs för kompilering av högra operanden. Då båda leden kompilerats kan proceduren avgöra om konvertering behövs. Om konvertering av vänstra operanden behövs så flyttar proceduren allt som genererats för högra operanden ett steg framåt och fyller därefter i konverteringen. Konverteringen hamnar då sitt rätta ställe, efter koden för vänstra operanden. Resultatet blir PUSH A%, Cvt%toF, PUSH B, AddFloat.

Listning av uttryck.

Ett krav vid specifikatitionen för kompilatorn är naturligtvis att alla uttryck skall gå att lista i den form de matades in. Det finns därför en rutin som översätter uttryck tillbaka från postfix notation till vanlig infix notation. Den klarar sig med den enligt ovan genererade koden med ett tillägg: Rutinen vet inte var parenteserna satt i det ursprungliga uttrycket. Om man skriver (5+5) eller 5+5 syns ju inte alls i den genererade koden. Kompilatorn lägger därför ut en kod för varje högerparentes som passeras. Koden ignoreras vid exekveringen men utnyttjas alltså vid LIST.

Kompilering av satser.

Kompileringen av satser drivs inte så långt som den av uttryck. Kompileringen inskränker sig till att verifiera att rätt ord kommer samt att byta dem till internkoder. En del adresser och avstånd beräknas också för att snabba upp exekveringen. Som ett exempel skall vi titta på satsen IF.

Vi antar att rutinen för kompilering av en sats har upptäckt att det står 'IF', genererat internkod för ordet och anropat IF-rutinen. IF-kompileringen börjar med ett anrop till uttryckskompilatorn för att ta hand om villkoret. Sedan kontrolleras om det händelsevis står 'THEN' efter uttrycket. Om det inte står THEN efter uttrycket är kompilatorn finkänslig nog att inte klaga utan jobbar vidare som om inget hänt. Efter detta hoppas en cell över som sedan fylls i med hur långt man skall hoppa om uttrycket var falskt (=0). Nu sker ett anrop rekursivt till satskompilatorn som alltså kompilerar vad som hända om uttrycket var sant. Efter detta anrop fylls cellen som just hoppades över i. Genom att använda avstånd och inte adress vinner man att cellen inte behöver ändras när hela IF-satsen flyttas omkring i minnet. Nu kontrolleras om det står 'ELSE' och i så fall sker en repris dvs en cell reserveras för ett avstånd. Satskompilatorn anropas och cellen fylls i.
Ett exempel behövs nog här:
IF A%=0% THEN PRINT ELSE STOP
genererar följande kod: (något förenklat)
/IF/, PUSH A%, PUSH 0%, =(integer), /THEN/, Skip x, /PRINT/, Skip y, /ELSE/, x:/STOP/, y:<radslut>.

Med /XXX/ menar jag här internkoden för XXX. Boktäverna x och y anger lägen i koden. (Jämför med labels i assembler!)

Listning av satser.

Eftersom kompileringen av satser inte är så långt driven är det också lätt att lista dem. Listrutinen byter bara de internkoder som motsvaras av BASIC-ord mot text och hoppar över övriga internkoder (Skip o dyl) och anropar baklängeskonvertering av uttryck då den stöter på en kod som hör till uttryck. Listrutinen vet alltså inte ens vilken typ av sats den håller på med.

5. Uttrycksberäknaren.

Uttrycksberäknaren anropas av de flesta rutiner för exekvering av satser när de behöver ett värde eller en pekare till en variabel. Uttrycksberäknaren i ABC80 är inte alls så stor och komplex som i de flesta andra BASIC-system. Eftersom uttrycken är i postfix notation och alla konverteringar är utlagda i internkoden kan uttrycksberäknaren gå igenom uttrycken helt blint utan att ta hänsyn till hur uttrycken ser ut. Beräknaren tar bara opkoderna i den ordning de uppträder och anropar en rutin för varje. Dessa rutiner gör alla någon operation med ett eller flera värden överst på stacken. Denna beräkningsstack är samma stack som CPU:ns hårdvarustack (SP).

Eftersom dessa rutiner modifierar toppen på stacken anropas dessa inte med CALL och avslutas inte med RET. Man kommer i stället in i dem med JMP och alla rutiner hoppar tillbaka direkt i beräknaren som bara hämtar nästa kod och fortsätter.

Det förekommer fyra datatyper på stacken under en uttrycksberäkning: Flyttal, heltal, strängar och adresser till variabler. Beräknaren vet aldrig vilka typer som ligger på stacken, hela ansvaret för vad som ligger där faller på kompilatorn. Varje rutin för opkoderna litar blint på att rätt typ ligger på rätt plats på stacken.

Det finns tio vägar ut ur beräknaren. Var och en av dessa har en egen opkod och även här har kompilatorn redan genererat rätt alternativ.

En av slutkoderna ger bara en RET tillbaka till anropet av beräknaren utan några data alls. Denna används endast sist i en DIM-sats och där finns ju inga data att överföra efter dimensioneringen. De återstående nio slutkoderna hänför sig till tre datatyper (Flyttal, Heltal och Strängar) kombinerade med följande tre grundtyper av resultat:

1. Slutresultatet kan vara ett värde. I det här fallet sker återhopp med kod som anger vilken datatyp det är och med värdet antingen på stacken eller i HL (om datatypen är heltal).

2. Slutresultatet kan vara en pekare till en variabel. Pekaren används till exempel vid READ eller INPUT för att tilldela en variabel ett värde. Återhopp sker med HL pekande på variabeln och kod som anger vilken variabeltyp som utpekas.

3. Uttrycket kan avslutas med en tilldelning. Detta gäller LET-satser och tilldelning av startvärde i FOR-satser. När beräknaren anropas med denna typ av uttryck sker tilldelningen och återhopp sker med kod som anger vilken variabeltyp som tilldelats. Pekare till variabeln fås även i detta fall.

Exekveringsrutiner

För varje sats i ABC80-BASIC finns en tillhörande exekveringsrutin. Dessa rutiner anropas från RUN-loopen (se kap 3).

Exekveringsrutinerna anropas med interpreteringpekare och de ansvarar för att pekaren pekar på första opkod efter satsen. Eftersom satser inte är så långt kompilerade fattas vissa beslut vid exekveringen, exempelvis vilken typ som skall skrivas ut i PRINT, eller till vilken typ inmatning skall konverteras vid INPUT eller READ.
Som exempel på exekveringsrutiner skall jag nedan beskriva PRINT och GOTO.

PRINT

Vid exekveringen av print kollas om 'PRINT' följs av ett nummertecken. I så fall anropas uttrycksberäknaren för att få vilken enhet som filversionen av PRINT skall gå till. Här kan man lugnt anta att enhetsnumret fås som heltal eftersom kompilatorn har lagt ut kod som säkert slutar med 'End of expr. integer value' (Se kap5).

Efter eventuell utvärdering av enhet följer en kontroll av om det inte finns mera i satsen. I så fall görs radframmatning och tillbakahopp till RUN-loopen. Om det finns något uttryck i satsen görs ett anrop av beräknaren för att få ett värde att skriva ut. Nu vet vi att kompilatorn har lagt ut kod som slutar med ett värde, men vi vet inte vilken typ värdet har eftersom alla typer skrivs ut med likadan PRINT-sats. Nu kommer returkoden från beräknaren väl till pass. Med hjälp av den kan vi avgöra vilken typ det nyss uträknade värdet fick.

Om värdet är heltal eller flyttal konverteras det till en ASCII-sträng med talet i läsbar form medan strängar naturligtvis inte behöver någon sådan konvertering. Nu skrivs värdet ut på bildskärmen eller filen och nästa post i PRINT-satsens lista undersöks.
Komma och semikolon specialbehandlas naturligtvis. Semikolon ignoreras helt om det inte står sist i listan då ju den avslutande radframmatningen uteblir.
Komma leder till att man kollar var på raden man befinner sig (på rätt enhet naturligtvis) och man beräknar det antal mellanslag som behövs för att man skall komma till nästa position delbar med 14. Om man skulle komma förbi högermarginalen görs i stället en vagnretur. Sist i kommahanteringen görs samma kontroll som vid semikolon, dvs så att radframmatning uteblir om kommat står sist i PRINT-satsens lista.

GOTO

GOTO-satsens exekveringsrutin är utan tvekan en av de enklaste. Man vet ju att efter GOTO följer alltid ett radnummer (redan kontrollerat vid kompileringen) och i samband med radnummer finns ju även adressen till den nya raden (ifyllt vid fixup, se RUN kap 3).

Det enda som behövs är att stega fram interpreteringspekaren till denna adress (3 steg) och ladda den med värdet den pekar på och sedan återvända till RUN-loopen.

Run-loopen kommer nu att börja på den nya raden utan att ens veta att hoppet överhuvudtaget har skett. GOTO-satsen sker alltså helt utan några beslut under exekveringen.

7. In- och Utmatning.

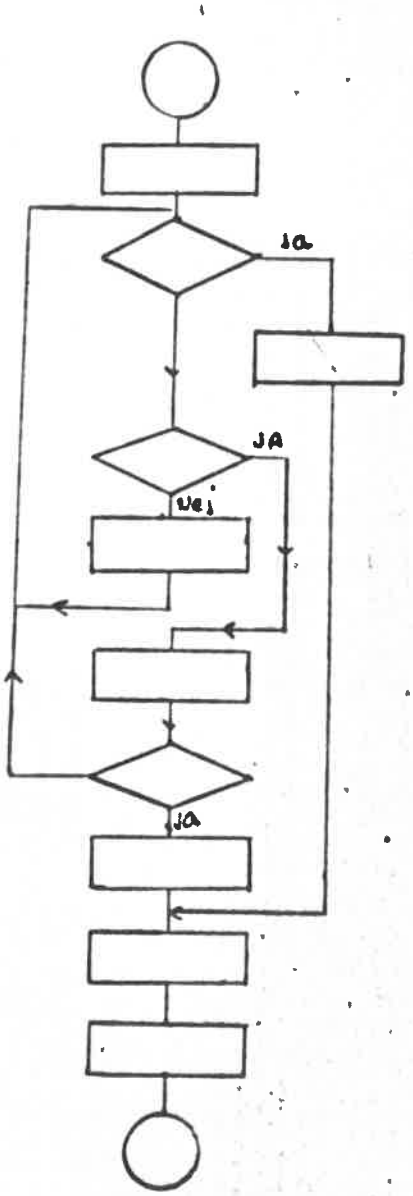
In och utmatningsrutinerna sönderfaller i två delar, dels mot konsolen, och dels mot övriga enheter. Konsolutmatning sker till det bildminne som finns på en fast adress (7C00 Hex). För att spara hårdvara ligger inte raderna konsekutivt i bildminnet utan omkastade på ett systematiskt sätt. Trots systematiken finns en tabell i ROM som talar om på vilken adress varje rad börjar. Programmet som driver skärmen (dvs skriver i minnet) simulerar en vanlig bildskärm, Lear Sieglers ADM 3A. Detta har drivits så långt att CUR-funktionen genererar ADM:s styrtecken för cursoradressering som sedan tolkas av skärmhanteraren. I stället hade man naturligtvis kunnat låta CUR-rutinen direkt modifiera skärmens rad- och kolumnadress, men då hade den bara fungerat mot konsolen. Med det lilla dubbelarbetet uppnår man att CUR-funktionen även fungerar mot andra bildskärmar av ovanstående typ, eller andra I/O-enheter där man också simulerar denna funktion.

Inmatningen är intressant. Här sker automatrepeat helt i mjukvara genom ett samarbete mellan interruptrutin och bakgrundsprogram. Man håller reda på en timer som kan laddas med två tider. Tiden T1 är tiden mellan första och andra tecknet som genereras av en enda men lång tangentnedtryckning. Detta är med andra ord tiden innan automatrepeaten sätter igång. T1 är ungefär 800 ms. Tiden T2 är tiden mellan tecknen när automatrepeaten väl har kommit igång. Denna tid är betydligt kortare, ungefär 120 ms.

Interruptrutinen behövs för att upptäcka de korta uppehåll i stroben från tangentbordet som fås av N-key rollover logiken då en ny tangent trycks ned (ca 5 mikrosekunder). Upptäckte man inte den nya tangenten skulle den av automatrepeatfunktionen felaktigt tolkas som en fortsättning av föregående tangentnedtryckning. Vid interrupt sätts en minnescell för att indikera det inträffade och dessutom initieras repeat-timer till T1 igen. Interruptrutinen behövs också för att under exekvering kunna kolla CTRL-C snabbast möjligt. Algoritmen för konsolinmatning med automat-repeat framgår av flödesshema.

Villkoret 'Ny tangent tryckt ?' kan testas på ovan nämnda minnescell. Villkoret 'Tangent fortfarande nere ?' testas genom att stroben på tangentbordsporten läses. PIO-porten måste därför köras i bitmod. (Se även diskussionen i initieringen, kap 1.)

Flödesplan konsolinmatning.



Start.

Beräkna adress på bildskärmen och tänd cursorblink.

Ny tangent tryckt ?

Nollställ 'nytt tecken-flaggan'. (Kvittera.)

Tangent fortfarande nere ?

Sätt timer till T1 (800 ms).

Minska timern med ett.

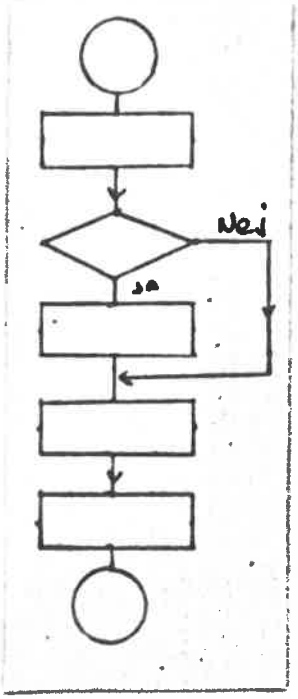
Blev den noll ?

Sätt timer till T2 (120 ms).

Läs in tangentbordets data från port.

Släck cursor-blink.

Slut.



Interrupt fås från keyboardets PIO.

Läs vilket tecken.

Var det CTRL-C ?

Sätt panikflaggan.

Sätt ' Nytt tecken'-flaggan.

Sätt timer till T1 (800 ms).

Hoppa ur interrupt.

7. In- och Utmatning.

In och utmatning till övrig periferiutrustning är gjord 'Device independent' dvs man behöver inte veta till vilken enhet in- resp utmatning skall ske när man skriver programmet. De behandlas lika. För varje typ av enhet finns därför en rutin för varje I/O-operation. Dessa anropas alltså lika oberoende av vilken enhet det är. Dessutom är I/O-hanteraren gjord så att det är lätt att addera flera enheter. Enhets-tabellen är nämligen gjord som en kedja där varje nod pekar på nästa, och vill man lägga till en ny går det lätt genom att lägga den först i kedjan.

Rutiner finns för dessa operationer.

1. Öppna befintlig fil.	Jfr BASIC-satsen OPEN.
2. Öppna ny fil.	Jfr PREPARE.
3. Stäng fil.	CLOSE.
4. Läs ASCII.	Används vid INPUT och LOAD av .BAS-filer.
5. Skriv ASCII.	Vid LIST och PRINT.
6. Läs recordvis.	Vid LOAD av .BAC-filer.
7. Skriv recordvis.	Vid SAVE.
8. Ta bort fil.	UNSAVE och KILL.
9. Döp om fil.	NAME A$ AS B$.

För printern (PR:) används bara 1,2,3 och 5. Övriga går till en dummyrutin som bara ger felet 'Illegal device operation'.

För IEC-bus används bara 1,2 och 3. Detta används till att styra signalen REN (Remote ENable) aktiv resp inaktiv. Själva in- resp utmatningen sker med specialsatserna CMD och IEC$(N$) som styr interfacet direkt.

För kassett-interfacet används alla operationer utom 8 och 9. 1 och 2 har fått en något annorlunda funktion.
1 (OPEN) betyder leta efter fil på kassett.
2 (PREPARE) betyder skriv filmärke på kassett.
OPEN används ju till öppning för läsning medan PREPARE används till öppning för skrivning.

För minifloppy är alla operationerna använda. Alla anrop görs till filhanteraren för minifloppy som ligger i PROM i floppyoptionen.

Filstrukturen på minifloppy är indexerad. Detta innebär att för varje fil finns en sektor som innehåller en tabell med pekare till alla sektorer som ingår i filen. Filen kan därmed ligga utspridd i små stycken var som helst på disken. Man slipper alltså allokera hela filen på en gång för att få ett tillräckligt stort kontinuerligt utrymme på disken.

Det finns en annan metod för dynamisk allokering av sektorer, länkad filstruktur där varje sektor innehåller pekare till nästa.

Indexerad filstruktur tillåter random access. Godtycklig sektor kan nås efter läsning av indexblocket. Länkad filstruktur (som inte finns i ABC80 men på många lågt stående system som MDS m fl) tillåter inte random access, här måste i värsta fall hela filen läsas för att man skall komma åt godtycklig sektor.

En finess finns i indexblocket och det är att flera sektorer som ligger i följd på disken förs samman till en post i indexblocket. Man kallar ett sådant sammanhängande stycke för ett segment omfattningen av ett segment kan alltså variera från en sektor till flera hela spår. Rutinen för allokering av nya sektorer till en fil tar också för sig ordentligt vid varje allokering annars skulle tabellen över lediga sektorer läsas onödigt många gånger. Skivan skulle också bli styckad i onödigt många små bitar.

Systemet med stora segment innebär att man kan få plats med flera sektorer i en fil utan att indexblocket blir fullt, och dessutom behöver man bara läsa om indexblocket vid övergång från ett segment till ett annat vid sekventiell läsning. (I en del system har man i stället index-blocken i minnet, men här sparar man minne utan att åtkomsten blir så värst mycket lidande.) Måttligt stora filer, och alla filer på en ny-initierad disk behöver ju bara bli ett segment och då går ju all åtkomst, såväl seventiell som random utan att man behöver läsa indexblocket mer än då filen öppnas.

Filstrukturen i ABC80 är helt kompatibel med den på Databoard 4680 minifloppy-system. Det är till största delen samma filhanterare med den skillnaden att ABC80:s är märkbart snabbare eftersom hela hanteraren ligger i ROM, dvs inga overlayer behöver laddas från floppyn.

Med floppyoption ansluten kan man även komma ur BASIC-systemet och komma in i diskoperativsystemet DOS. Detta sker genom att man ger kommandot BYE. Då laddas filen CMDINT.SYS in i RAM-minnet och detta är samma kommandoavkodare som finns i Databoard 4680 DOS. Under ABC80-DOS kan man sedan köra i princip alla program efter omassemblering med den självklara begränsningen att de måste få plats i minnet (16 kbytes).

Med kommandot $BAS kommer man tillbaka till ABC80 BASIC.

Verktygen - hur jobbet gått till praktiskt.

Programvaran är utvecklad hos Dataindustrier i Täby. Där används en Interdata 7/32 som arbetsdator att köra editering och assemblering på.

Utan goda verktyg såsom rejäl dator med någorlunda stor disk och snabb radskrivare skulle ett så här stort program knappast kunna utvecklas. Man skulle antagligen tröttna på alla dåliga verktyg innan jobbet var i det närmaste klart.

Andra oumbärliga hjälpmedel är en bra relokerbar assembler och länkare. Med dessa har man möjlighet att modularisera sitt program så att man inte behöver assemblera allt då man ändrar i en enda subrutin. Invar Larsson på Symtek har skrivit en assembler som uppfyller det mesta man kan önska sig av finesser.

Den är inte skriven för ZILOG:s egna mnemonics. Dessa avvisades på ett tidigt stadium eftersom de var lite för svåra att skilja åt. (De allra flesta instruktioner heter LD.) Dessutom eftersträvas en viss standardisering så att samma operation heter likadant antingen den skall exekveras i en IBM, Interdata, Intel 8080 eller Z80.

Bilaga 1: Några användbara adresser i ROM.
Bilaga 2: Exempel på lista genererad med Ingvars assembler.
Bilaga 3: Manual (från Scandia Metric).

Några adresser i ROM som kan vara användbara då man skriver assemblerprogram.

0000H START Omstart av BASIC i ABC80.

0002H CONSI Hämtar ett tecken från tangentbord till A.
Jfr GET.

0005H RDCONS Hämtar en rad från tangentbord inklusive
eko och back-space processning.
Anropa med:
Pekare till var man vill ha strängen i HL.
Max längd i C.

000BH WRCONS Skriver på skärmen.
Anropa med:
Pekare till data att skriva i HL.
Antal bytes i BC.

```
 PLC      B   KOD       ARG      LC   STATEMENT----------------------------------

 000 000                          1   UNARY     PROG  COMPILE UNARY OPERATORS
                                  2   *
                                  3   *         HANDLE UNARY OPERATORS + AND -
                                  4   *
 000 000      176                 5   UNPLMI*   L     A,(H)                 GET A BYTE
 000 001      376 055             6             CI    '-'
 000 003      050 010             7             JZS   WASMIN
 000 005      376 053             8             CI    '+'
U000 007    X 314 000 000         9             CZ    SKIPSP                JUST SKIP PLUS
U000 012    X 303 000 000        10             J     BINARY
                                 11   *
 000 015    X 315 000 000        12   WASMIN    CALL  SKIPSP
 000 020      335 345            13             PUSH  X
U000 022    X 315 000 000        14             CALL  NEXTTYP               GET THINGS TO NEGATE
 000 025      335 341            15             POP   X
 000 027      330                16             RC
U000 030    X 076 000            17             LI    A,INTTYP
 000 032      270                18             CR    B
U000 033    X 076 000            19             LI    A,NEGINTOP
 000 035      050 011            20             JZS   WASM1
L000 037    X 076 000            21             LI    A,FLTTYP
 000 041      270                22             CR    B
U000 042    X 076 000            23             LI    A,E.MODERR
 000 044      067                24             STC
 000 045      300                25             RNZ                         DON'T NEGATE STRINGS !
U000 046    X 076 000            26             LI    A,NEGFLTOP
 000 050      022                27   WASM1     ST    A,(D)
 000 051      023                28             INCD  D
U000 052    X 303 000 000        29             J     NXTEIN                PROCEED AS IF BINARY
                                 30   *
 000 055                         31             EJECT
```

```
 PLC      B   KCD        APG     LC   STATEMENT----------------------------------

                                  32   *
                                  33   *          HANDLE NOT
                                  34   *
  000 055      325               35   NOTTYP*    PUSH   D
U 000 056    X 335 136 000       36              L      E,BIN.RW(X)          GET POINTER TO 'NOT'
  000 061    X 335 126 001       37              L      D,BIN.RW+1(X)
U 000 064    X 315 000 000       38              CALL   LEXSCAN
  000 067      321               39              POP    D
  000 070    X 302 000 000       40              JNZ    NEXTTYP              WAS NO 'NOT'
  000 073    X 315 000 000       41              CALL   NEXTTYP              GET WHAT TO COMPLEMENT
  000 076      330               42              RC
L 000 077    X 315 000 000       43              CALL   MAKEINT              MUST BE INT.
  000 102      330               44              RC
U 000 103    X 076 000           45              LI     A,NOTRW
  000 105      022               46              ST     A,(D)
  000 106      023               47              INCD   D
  000 107      311               48              RET
                                  49   *
  000 110                         50              END
ERRORS:    0     WARNINGS:    13
```

PLC	B	KOD	ARG	LC	STATEMENT
U<BIN.RW		000 000	36	37	
U<BINARY		000 000	10		
U<E.MODERR		000 000	23		
U<FLTTYP		000 000	21		
U<INTTYP		000 000	17		
U<LEXSCAN		000 000	38		
U<MAKEINT		000 000	43		
U<NEGFLTOP		000 000	26		
U<NEGINTOP		000 000	19		
U<NEXTTYP		000 000	14	40 41	
U<NOTRW		000 000	45		
>NOTTYP		000 055	35		
L<NXTBIN		000 000	29		
U<SKIPSP		000 000	9	12	
>UNPLMI		000 000	5		
WASM1		000 050	27	20	
WASXIN		000 015	12	7	