Datorteknik
Morselabben i FORTH

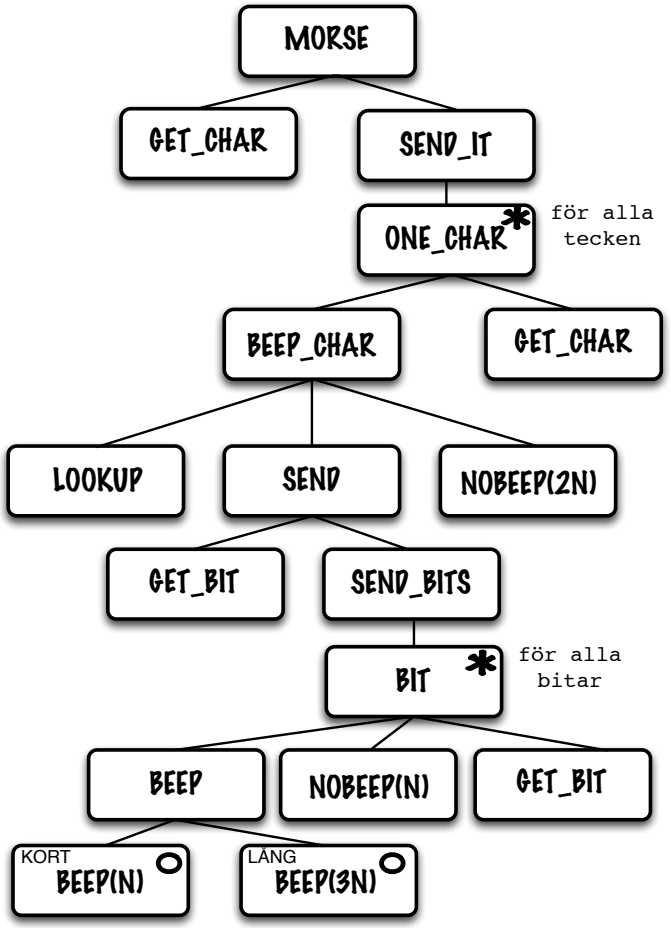
Michael Josefsson

31 oktober 2020

Inledning Utan att avslöja för mycket om hur morselaborationen (Lab2) utförs i assembler kan vi betrakta dess välstrukturerade implementation i programspråket **FORTH**.

Huvudsyftet med denna text är att påvisa *tänket* med att allt är subrutiner. Kan den dessutom väcka ett intresse för det något udda, men ack så eleganta, programspråket **FORTH** är det bara så mycket bättre. Det är nyttigt att vara bekant med flera programspråk.

Labhäftet anger JSP-strukturdiagram och pseudokod med dessa bilder:




```
MORSE{
      CALL GET_CHAR;
      while( char != 0){
         CALL LOOKUP;
         CALL SEND;
         CALL NOBEEP(2N);
         CALL GET_CHAR;
      }
}
```

```
SEND{
      CALL GET_BIT;
      while ( Z != 0){
           if bit==0
                CALL BEEP(N);
           else
                CALL BEEP(3N);
           CALL NOBEEP(N);
           CALL GET_BIT;
      }
}
```

Utifrån denna pseudokod kan nu kod skrivas för vilket programspråk som helst. I Lab2 översätts underlaget exempelvis till AVR-assemblerkod men i och med att problemet är nedbrutet till pseudokod är även andra programspråk möjliga.

I strukturdiagrammet är uppgiften uppdelad i åtskilliga små rutiner. Dessa rutiner utför uppgifter som måste finnas för ett komplett program. Typiskt motsvarar en ruta i strukturdiagrammet en subrutin i assembler eller en funktion i något annat språk.

Att slaviskt konstruera programmet enligt strukturdiagrammet ger dock en hel massa subrutiner. Rutiner som kan vara svåra att hålla reda och, inte minst, hitta bra namn till. I pseudokoden ovan till höger har JSP-strukturdiagrammet översatts till större enheter och därmed fått en kanske mer hanterbar uppdelning centrerad kring var sin uppslagning i en tabell: `MORSE` letar upp nästa tecken som skall sändas och `SEND` letar upp nästa teckendel och sänder dessa.

FORTH Programspråket är udda men är egendomligt nog ett av de mest spridda. Det tar inte lång tid förrän en ny processortyp får sin första **FORTH**-implementation. En av anledningarna är att det är relativt enkelt att porta en befintlig **FORTH** till ny hårdvara.

FORTH är mycket strukturerat uppbyggt där senare delar av språket byggs av tidigare definierade delar. Någon annan uppdelning är inte möjlig. I extremfall räcker det att ett dussin **FORTH**-*ord* översätts till assembler så är implementationen klar då språkets senare delar alla beror på dessa första definitioner.[1]

Med en fungerande implementation kan man via tangentbordet *direkt* kommunicera med det nya systemet. Man behöver alltså inte skriva kod i en fil och sedan kompilera och ladda ner till målsystemet. Allt detta sker direkt när man slår på retur-tangenten.

Det finns mycket beskrivningar av **FORTH** på nätet[2] så den intresserade kan hitta detaljerad information där. För vår del behöver vi känna till att hela språket byggs upp av *ord* och att programmeringen handlar om att utöka den den befintliga *vokabulären* med egna ord som gör det önskade. Ett ord definieras med ett inledande ":" och avslutas med ";".

Något annat vi behöver känna till, och som är centralt i språket, är att det knappt använder variabler utan förlitar sig på en parameterstack för argument och beräkningar.

Stacken är en vanlig datastruktur men mer sällan används den på ett så direkt sätt som i **FORTH**: All text som inte kan hittas som definierade ord går vidare till ett ord, **number**, som försöker översätta texten till en siffra, givet nuvarande bas (som återfinns i variabeln **base**, även detta ett *ord*) och, om den lyckas, sedan lägger siffran på stacken.

[1] Ok, det är kanske *liiite* förenklat men i princip går det till så.
[2] För arduinokortet är *FlashForth* mycket lämplig.

Alla siffror man matar in läggs alltså på stacken och kan sedan manipuleras, exempelvis utför man additionen `13 + 16` genom att lägga termerna på stacken och utföra additionen med ordet `+`:

```
input             | stack
------------------------
13                | 13
16                | 13 16
+                 | 29
```

Resultatet kan sedan skrivas ut till skärmen med ordet `.` varvid stacken töms. Vill man ha kvar resultatet kan det dupliceras med ordet `dup` innan utskrift:

```
input             | stack
------------------------
                  | 29
dup               | 29 29
.                 | 29
```

Flera små (så kallade *primitiver*) finns för att manipulera stacken på detta sätt, bland annat `swap`, `over` och `drop` vars stackeffekter visas nedan

```
input             | stack
------------------------
13                | 13
16                | 13 16
swap              | 16 13
over              | 16 13 16
drop              | 16 13
```

Denna stackmanipulering tar ett tag att få grepp om och det är onekligen en anledning till att språket betraktas som udda. Det är dock ett väldigt snabbt och resurssnålt sätt att hantera variabler och argument.

Ett ord, låt oss kalla det `1+`, som adderar konstanten `1` till vad som finns på stacken kan nu definieras som

```
: 1+    1 + ;
```

och vi ser att ordet består av det tidigare definierade ordet och siffran `1`.

För att hålla ordning på ett ords stackeffekt brukar man dokumentera enligt

```
: 1+    1 + ;    ( n --- n+1 )
```

Där talet `n` ligger på stacken innan `1+` exekveras och så klart `n+1` efter.

Naturligtvis innehåller språket både variabler och konstanter såväl som kontrollstrukturer, `if`-satser med flera, och så vidare. De dyker upp i programmet nedan med kommentarer för att underlätta läsningen.

Morselabben En implementation av laborationen i **FORTH** är denna.

```
$24   constant DDRB            \ define I/O-adress for DDRB
$25   constant PORTB           \ define I/O-adress for PORTB
$50   constant length          \ tone length for 'dit'
$100  constant pitch           \ frequency of audio

: delay pitch for next ;       \ loop 'pitch' rounds

: beep ( length flag --- )     \ beep for 'length' if flag true
     swap                      \ flag length ---
     for                       \ iterate length times
        dup PORTB c! delay     \ output flag and delay
        0 PORTB c! delay       \ output zero and delay
     next drop                 \ remove flag to clean stack
;

hex                            \ now use hexadecimal base for input

flash create mtab              \ make table 'mtab' in flash
60 c, 88 c, a8 c, 90 c, 40 c, 28 c, \ c, stores one byte
d0 c, 08 c, 20 c, 78 c, b0 c, 48 c,
e0 c, a0 c, f0 c, 68 c, d8 c, 50 c,
10 c, c0 c, 30 c, 18 c, 70 c, 98 c,
b8 c, c8 c,

decimal                              \ back to decimal base

: ditorda ( flag --- )               \ calculate length of tone, 1T|3T
  length dup rot                     \ length length flag
  if 3 * then                        \ length 1T|3T
  -1 beep                            \ length true beep
  0 beep                             \ 1T silence
;

: asciisend ( ascii ---)
  65  - mtab + c@                    \ convert to mtab-index and lookup
  dup
  if                                 \ for each bit from mtab-code
    begin
        dup 128 and 0= 0= ditorda    \ send dit or dah
        2* dup 127 and  0=
    until drop
  then
  length 2* 0 beep                   \ 2T silence between chars

;
```

```
: str s" DATORTEKNIK "  ;          \ string to send

: morse
  $ff DDRB c!                     \ PORTB is output
   begin
      str 1-                      \ zeroindex
      for                         \ treat each char in turn
        dup c@ asciisend 1+
      next
    drop drop                     \ clean stack
  again                           \ forever

;
```

Notera hur referenser till senare del av koden inte förekommer och inte heller **kan** förekomma. Allt byggs upp i små funktionella enheter från programmets start.

Varför FORTH? Om man bortser från det uppenbara *Because it is there!* kan man notera att assemblerkod med nödvändighet innehåller en hel del detaljer, detaljer som belastar programmeraren. För att avlasta hjärnan alla dessa detaljer måste vi blocka, aggregera, information i större grupper eller funktioner.

FORTH tillåter en beskrivning som ligger nära assembler i själva tänket men ändå förenklar programmeringen då språket innehåller till exempel kontrollstrukturer, variabler och konstanter.

Detta måste upplevas för att sjunka in. En interaktiv *live*-session i **FORTH**, där man direkt kan skriva till en port för att testa hårdvaran och konfigurationer (**$80 PORTB c!**) är mycket praktiskt och när man vant sig vid denna omedelbara återkoppling saknar man den starkt vid ”vanlig” programmering.

Med sådana korta testrader eller testord kan man konstatera att hårdvaran beter sig som avsett och sedan inkludera dem i det stora programmet.

En annan, kanske inte lika uppenbar fördel, är att språket **helt omöjliggör** ostrukturerade lösningar. Det finns helt enkelt ingen möjlighet att vildhoppa hit och dit. Programmet måste följa en struktuerad gång vilket tränar programmeraren att tänka på ett modulärt och strukturerat sätt. Kan man bryta ner sitt problem till en implementation i **FORTH** har man tillägnat sig ett viktigt programmeringstänk.

—o=Ö=o—