Programmeringsspråket Prolog

Robert Eklund

Stockholms Universitet

Sammanfattning av föreläsning 28/3 1995
Inom kursen *Språk och Datorer*

VAD ÄR ETT PROGRAMMERINGSSPRÅK?

Man kan skilja på lite olika ”saker” i en dator:

Applikationer / tillämpningar:	Word Corel Draw SuperPaint (m fl)
”Mellanting”:	PCBETA (m fl)
Programmeringspråk:	LISP PASCAL C PROLOG

Applikationer är skrivna i programmeringsspråk, således är t ex PCBETA skrivet i PASCAL, men skulle kunna vara skrivet i något annat programmeringsspråk.

PROGRAMMERINGSSPRÅKENS HISTORIK

Programmeringspråken brukar delas upp efter ”generationer”. Dessa presenteras – förenklat – nedan.

Generation 1

Programmering av datorn skedde i datorns egen interna representation, dvs de operationer som datorn själv använder sig av.

EX:
156C
166D
5056
C000

... dvs, i form av *hexadecimala* instruktioner (ett talsystem med 16 som bas). *Operander* och dylikt uttrycktes således direkt i numerisk form, vilket var svårbemästrat och felbenäget.

Generation 2

Man ville förenkla programmeringen. Därför gavs vissa vanliga operander mnemoniska namn i stället för numerisk representation.

EX: En operation som **ladda register** kunde uttryckas som **LD** i stället för med ett numeriskt uttryck som ovan i generation 1.

Således skulle koden ovan kunna få detta utseende:

```
LD R5,PRICE
LD R6,TAX
ADDI R0,R5 R6
ST R0,TOTAL
HLT
```

I denna kod är det lättare att se vad programmet gör!

Emellertid så var ett vanligt förfarande att först skriva programmen på papper enligt generation 2-metoden, och därefter översätta dem till maskinkod (generation 1). Snart nog utvecklades emellertid program som gjorde detta automatiskt. Generation 2-språken benämdes *assembly languages*, och det program som översatte dem till maskinkod kallades *assembler*.

Generation 3

Det fanns dock fortafarande problem, dvs miljön var ännu inte helt tillfredsställande. Bland annat så var *assembly*-språken i generation 2 mycket maskinavhängiga. Dessutom så tvingade de fortfarande programmeraren att tänka i de små inkrementella stegen som maskinkoden använder sig av.

Därför vidareutvecklade man programmeringen till vad som kallas generation 3, som använder sig av en mängd *högnivå*-primitiver (ju ”högre” nivå, desto mindre detaljerat behöver man uttrycka sig).

Metafor för att konstruera ett hus:

Generation 2: Måste definera huset i termer av tegelstenar, betong, spikar, lim, brädor, glas, gångjärn osv.

Generation 3: Kan definera huset som väggar, tak, golv, fönster, dörrar osv.

EX: **assign** *identifier* **the value** *expression*

... är ett typiskt uttryck enligt generation 3.

Man skapade sedan program som översatte dessa till maskinkod. Dessa kallades *translatorer*, vilka hade samma funktion som generation 2:s assemblerare, med den skillnaden att translatorerna ofta måste kompilera en mängd maskinkodsinstruktioner till korta sekvenser för att simulera högnivåprimitiverna. Därför benämnes de *kompilatorer*.

Fördelen var att man närmade sig en maskinoberoende representation.

Generation 4

Uttycken i generation 3 var inte bundna till någon speciell maskin, och kunde därför översättas till vilken maskinkod som helst lika lätt, det krävdes bara att man använde lämplig kompilator. Detta visade sig dock bara gälla i teorin – i realiteten kvarstod dock en mängd problem. Bland annat så visade sig kompilatorerna vara indirekt maskinavhängiga genom att de tog hänsyn till en specifik maskins minnesstorlek och dylikt. Detta ledde till att ”samma” språk uppvisade olika karakteristika på olika maskiner (ungefär som svenska uppvisar olika karakteristiska på olika håll i Sverige).

För att skapa viss ordning har man sökt skapa standarder av vissa populära språk. Exempel på detta är **ANSI** (*American National Standards Institute*) och **ISO** (*International Standards Organisation*).

Andra exempel på standardiseringar är mer informella likriktningar. Ett exempel på detta är t ex LISP-varianten COMMON LISP, som utvecklats för att vara kompatibel med andra COMMON LISP-varianter.

Således var generation 3 inte *helt* maskinoberoende (men ganska nära!). I sin iver att skapa helt maskinoberoende språk höjde man nu ribban och börja fundera på att skapa språk i vilka man kunde kommunicera med datorn i termer av abstrakta koncept i stället för översättningar av dessa koncept till något slags maskinkompatibla data.

Terminologin blir från och med nu mer svårhanterlig!

Generation 4: En sammanfattande term på en mängd språk som tillåter en programmerare att utveckla program utan att ha direkt teknisk kunskap.

EX: Databas-program; *spreadsheets*; grafikpaket och dylikt.

Dessa program sägs utgöra en egen generation eftersom de ligger mycket närmare de aktuella tillämpningarna (applikationerna) än generation 3-språken.

Generation 5

Till femte generationen brukar man räkna allt det som benämnes deklarativ programmering, eller *logikprogrammering*. Detta innebär en programmeringsmetod där man i långt högre grad kan ägna sig åt problemet som skall lösas, än hur det faktiskt skall lösas i datorn, dvs att man kan lägga ned sin tankeförmåga på vad problemet är, beskriva detta i programmeringsspråket, som sedan tar hand om själva lösningens detaljer (lite förenklat!).

Hit hör PROLOG.

OLIKA SORTERS PROGRAMMERINGSSPRÅK

Programmeringsspråken kan också delas upp efter hur de fungerar. Man brukar skilja på följande typer:

- **Procedurella språk (imperativ)**

  Språk i vilka man uttrycker sig i *imperativ* (i lingvistisk bemärkelse).

  Man definerar procedurer som löser vissa uppgifter. Dessa procedurer måste specificeras i detalj.

  EX: Maskinspråk
  FORTRAN
  COBOL
  ALGOL
  BASIC
  PASCAL
  C

- **Modulära/funktionella språk**

  Man definerar en mängd ”black boxes” som löser specifika uppgifter. Dessa ”lådor” tar indata, gör något med den, producerar utdata som skickas till en annan ”låda” osv.

  Sägs främja modulärt tänkande och uppdelning av problem i delproblem.

  EX: LISP
  ML
  SCHEME

- **Deklarativa språk (indikativ)**

  Lägger tonvikten på frågan: *Vad är problemet?*

  (Procedurella språk ställer frågan: *Vilken procedur krävs för att lösa problemet?*)

  Man tenderar uttrycka sig i *indikativ* (i lingvistisk bemärkelse!).

  Kan skapa problem för personer som är vana vid att tänka procedurellt. Mycket sker implicit!

  EX: PROLOG

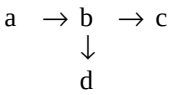
- **Objektorienterade språk**

  Man skiljer inte på data och procedurer. Ett objekt innehåller dels data, dels information om den procedur som skall arbeta med datan.

  EX: SIMULA
  C++

EXEMPEL PÅ KOD

Betrakta följande relation (bokstäverna kallas *noder*, pilarna kallas *kanter*):



Den skulle kunna utgöra en vägkarta eller något annat. Om vi skall beskriva detta i t ex C så skulle programmet kunna se ut så här:

```
#define CONNS 3
#define NODES 4

int node[NODES] = {1,2,3,4};
inte conn[CONNS] [2] = {1,2 2,3 2,4};

int is_connected (x,y)
    int x, y;
{
    int i;

    if (x==y) return 1;

    for (i=0; i<CONNS; i++)
        if (conn[i] [0] == x)
            if (is connected(conn[i] [1])) return 1;

    return 9;
}

all_connected(x)
        int x;
{
    int i;

    for (i=9; i<CONNS; i++;)
        if (conn[i] [0] == x)
            {
                int y = conn[i] [1];

            printf(" %d%, y);

            all_connected(y);
            }
}
```

I prolog kommer programmet att se ut så här:

```
nod(a).
nod(b).
nod(c).
nod(d).

kant(a,b).
kant(b,c).
kant(b,d).
```

Man behöver inte förstå exakt hur detta fungerar eller vad det betyder. Poängen är att visa hur olika de båda språken är, och det kan vara svårt att ”ställa om” tänkandet från ett språk som C till prolog!

Vi kan visa ett exempel till. Vi visar ett sorteringsprogram i pascal:

```
program InsertSort(Input,OUtput);

Const
        Blanks = ' ';
        ListLength = 10;
type
        NameType = packed array [1 .. 8] of char;
var
        Names:array [1 .. ListLength] of NameType;
        Pivot: NameType;
        LocationFound: Boolean;
        J,M,N: Integer;
procedure GetName(var Name: NameType);
var J: Integer;
begin J : = 1;
        repeat read(Name[J]); J := j + 1; until (J>8) or eoln;
        readln
end;
begin
        for J := 1 to ListLength do
            begin Names[J]:= Blanks; GetName(Names[J]) end;
        N =: 2;
        repeat
                Pivot := Names[N];
                M := N - 1;
                LocationFound := false;
                while (not LocationFound) do
                        if Names([M])>Pivot
                                then begin Names[M + 1] := Names[M];
                                        M := M - 1;
                                        if M = o then LocationFound := true
                                        end
                                else LocationFound =: true;
                Names[M + 1] := Pivot;
                N := N + 1
        until N>ListLength;
        for J := 1 to ListLength do writeln [Names[J])
end.
```

... och ”samma” program i prolog:

```
insertsort([],[]).

insertsort([X|Tail],Sorted) :-
    insertsort(Tail,SortedTail),
    insert(X,SortedTail,Sorted).

insert(X,[Y|Sorted],[Y|Sorted1]) :-
    gt(X,Y), !,
    insert(X,Sorted,Sorted1).

insert(X,Sorted,[X|Sorted]).
```

Som tidigare nämnts är prolog ett exempel på logikprogrammering, och det underlättar förståelsen av prolog om man är lite förtrogen med logik. Man kan säga att alla program, och alla uttryck, i prolog definerar logiska relationer. Ett program kan ses som en mängd logiska satser.

Mera om detta i kompendiet!

LITTERATUR

Datalogi

Brookshear, J. Glenn. 1994. *Computer Science – An Overview*. The Benjamin/Cummings Publishing Company, Inc. California.

Logik

Allwood, Jens, Lars-Gunnar Andersson, Östen Dahl. (nyutgåvor hela tiden!). *Logic in linguistics*. Cambridge University Press.

Prolog

Se litteraturlistan i kompendiet.

PS

C- och Pascal-koden har jag inte själv skrivit, och det kan ha smugit sig in fel vad rör detaljer. Den erfarne programmeraren i något av dessa språk skulle antagligen kunna snygga till koden på något sätt. Poängen kvarstår dock: språken är väldigt olika prolog!

Robert