Prova på-laboration i Ada

Peter Dalenius
petda@ida.liu.se
Johan Sjöholm
johsj@ida.liu.se
Institutionen för datavetenskap, Linköpings universitet
2009-07-28

1. Introduktion till imperativa språk

Programspråket Ada är resultatet av ett stort utredningsarbete under senare delen av 1970-talet. Det hela tog sin början 1974 när det amerikanska försvarsdepartementet *Department of Defence* initierade ett arbete med att ta fram ett nytt gemensamt högnivåspråk för att ersätta de inte mindre än 450 olika språk som då användes i olika försvarsapplikationer. Kraven på det nya språket genomgick en lång och tungrodd process och först 1983 kom den första officiella standarden av Ada kallad Ada83. Eftersom många av försvarets datorsystem var inbyggda, dvs datorer monterade i t.ex. stridsvagnar eller robotar, var två viktiga krav att språket skulle vara stabilt och uppdelat i moduler. Detta är något som man ser tydliga spår av i språket även idag.

Ada är ett *imperativt* språk. Från grundskolegrammatiken kanske man minns att imperativ är den verbform som utgör uppmaningar och det är precis vad det handlar om även när det gäller programmering. Program skrivna i imperativa språk innehåller uppmaningar som talar om för datorn vad den ska göra. Gruppen av imperativa programmeringsspråk är stor. FORTRAN, BASIC, Pascal, C och Ada är exempel på välkända och framgångsrika programspråk som är imperativa.

Det finns tre viktiga progammeringselement som man kan hitta i framför allt imperativa språk: sekvens, villkor och upprepning.

Sekvens

Om man ska utföra flera saker efter varandra ordnar man dem i en *sekvens*. Det låter kanske så naturligt att det känns fånigt, men det är långt ifrån självklart för alla programspråk. När det gäller imperativa språk är dock sekvenser av instruktioner en av hörnstenarna. Ofta placerar man en instruktion eller *sats* per rad för att vara extra tydlig. Ett exempel i vanligt språk skulle kunna vara:

```
MORGONRUTIN
Kliv upp ur sängen
Ta en dusch
Klä på dig kläderna
Ät frukost
Borsta tänderna
Gå till jobbet
```

Man börjar uppifrån och utför helt enkelt instruktionerna en rad i taget.

Villkor

I många program hamnar man i en situation där man vill göra endera av två saker. Dessa kallar man för *val*, *villkor* eller *selektion*. Det grundläggande sättet att hantera valsituationer i de flesta språk, inte bara imperativa, är att ha en så kallad `if`-sats. Ett exempel som blandar Ada och vanligt språk är:

```
RUTIN FÖR ATT ÄTA FRUKOST
if Det finns flingor then
  Ät flingor
else
  Ät smörgås
end if
```

Man inleder `if`-satsen med att testa om ett visst villkor är uppfyllt. Om så är fallet utförs det som står på nästa rad, efter `then`. I annat fall utförs det som står efter `else`. För att vara extra tydlig brukar man skriva det som ska utföras i de olika fallen lite längre in på raden. Man säger att man *indenterar* dessa rader. Man kan också vid behov hoppa över `else`-grenen.

Upprepning

Ofta vill man upprepa saker i program. Ibland kanske man vill göra något ett bestämt antal gånger, andra gånger vill man göra något tills ett visst villkor är uppfyllt. Upprepning brukar också kallas för *iteration* och det ordnar man i många programspråk med så kallade *loopar*. Exempel:

```
RUTIN FÖR ATT ÄTA FLINGOR
while Skålen med flingor inte är tom loop
  Ta en sked flingor
  Tugga
  Svälj
end loop
Ställ undan disken
```

Ovanstående låtsaskod är ett exempel på en `while`-loop där man upprepar tre satser så länge skålen med flingor inte är tom. För att upprepa något ett bestämt antal gånger använder man i många språk en `for`-loop istället.

```
RUTIN FÖR ATT ÄTA SMÖRGÅS
for I in 1..3 loop
  Bred smör på en smörgås
  Lägg ost på smörgåsen
  Ät smörgåsen
end loop
Ställ undan smöret och osten
```

I en `for`-loop har vi alltid en *loop-variabel* som hjälper oss att hålla ordning på vilket varv vi är på. Första varvet kommer I att ha värdet 1, andra gången 2 osv. En av egenheterna med Ada är att loop-variabler inte behöver deklareras.

2. Exempel på Ada-program

Ett Ada-program består i princip av tre olika fasta delar enligt nedanstående mall:

```
-- Vilka paket vill man använda?
procedure Mitt_Program is
  -- Vilka variabler vill man använda?
begin
  -- Vad vill man utföra?
end Mitt_Program;
```

Alla rader som inleds med -- är kommentarer. Först i programmet talar man om vilka *paket* man vill använda. Språket Ada är nästan helt och hållet uppbyggt av olika paket som t.ex. kan innhålla rutiner för att skriva ut saker eller kommunicera via nätverk. Om man inte inkluderar några paket kommer programmet knappt att kunna göra något alls.

Programmet egentliga början kommer på raden med `procedure`. Här talar man om vad programmet heter. Efter denna rad *deklarerar* man sina variabler, dvs man talar om vilken information man behöver lagra under programmets gång. Mellan `begin` och `end` följer sedan själva programmet, dvs satserna som talar om vad som ska hända. Följande exempel visar ett komplett Ada-program.

```
-- Paket som ska användas
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

procedure Summera_Tal is

   -- Deklaration av variabler
   Antal_Tal : Integer;
   Summa : Integer;
   Tal : Integer;

begin
   -- Själva programmet

   Put("Ange antal tal: ");
   Get(Antal_Tal);

   Summa := 0;
   for I in 1..Antal_Tal loop
      Put("Mata in tal: ");
      Get(Tal);
      Summa := Summa + Tal;
   end loop;

   Put("Summan blev: ");
   Put(Summa);
   New_Line;

end Summera_Tal;
```

De första två raderna talar om att vi vill använda två paket som heter `Ada.Text_IO` och `Ada.Integer_Text_IO`. Dessa innehåller rutiner för att läsa in och skriva ut text och heltal. De gör att vi kan använda procedurerna `Put`, `Get` och `New_Line` senare i programmet.

Programmet i exemplet heter `Summera_Tal`. Alla namn på program, procedurer eller variabler i Ada skrivs med stor bokstav i början och små bokstäver därefter. Om ett namn består av flera delar sätts de ihop med *underscore*. Detta är enbart en praxis. Ada-kompilatorn klarar av att hantera variabler med små bokstäver, men en enhetlig kodningsstil gör programmen lätta att läsa.

Vi deklarerar hur som helst tre variabler som heter `Antal_Tal`, `Summa` och `Tal`. Dessa är samtliga av typen `Integer`, dvs heltal. Varje deklaration avslutas med `;` (semikolon).

Själva programmet består av en sekvens av satser som utförst från början till slut. Varje sats avslutas med semikolon och man skriver normalt en sats per rad. De två första satserna skriver först ut en text på skärmen med `Put` och läser därefter in ett heltal till variabeln `Antal_Tal` med `Get`. Dessa två satser är *proceduranrop*, dvs de anropar ett underprogram som finns någon annanstans (i detta fallet i något av paketen nämnda i början). Proceduranrop består, som i många andra språk, av namnet på proceduren följt av ett eller flera *argument* inom parentes.

För att göra programmet mer lättläst finns några tomma rader inskjutna då och då. I nästa del av programmet kommer en *tilldelningssats*. Den innebär att det som står till höger om tilldelningsoperatorn `:=` beräknas och resultatet placeras i variablen som står till vänster. I det här fallet placeras alltså talet 0 i variablen `Summa`. Därefter sker en upprepning med en `for`-loop. Loop-variablen I används och den får i tur och ordning värdena 1, 2, 3, ... upp till det värde som finns i variabeln `Antal_Tal`. Som vi nämnt tidigare behöver inte loop-variabler i Ada deklareras. Det som sker i varje varv av upprepningen är att en text skrivs ut på skärmen, ett tal läses in till variabeln `Tal` och att `Summa` uppdateras så att `Tal` läggs till det tidigare värdet.

I det sista avsnittet av programmet skrivs en förklarande text ut. Därefter skrivs summan ut. Till sist sker en radmatning med proceduren `New_Line` som inte tar några argument.

3. Att använda Ada

Ada är ett *kompilerande* språk. Det innebär att *källkoden*, dvs programmets ”text” som är läsbar för människor, måste översätts i ett enda svep till ett riktigt körbart program. Vi använder en kompilator som heter GNAT[1] och för att den ska fungera måste exempelprogrammet sparas i en fil med namnet `summera_tal.adb`, dvs samma namn som programmet fast med enbart små bokstäver. För att kompilera använder man skalkommandot `gnatmake`. Exempel:

```
zaza3 <12> gnatmake summera_tal
gcc -c summera_tal.adb
gnatbind -x summera_tal.ali
gnatlink summera_tal.ali
zaza3 <13>
```

Detta kommer att producera en körbar fil `summera_tal`, men också ett par skräpfiler `summera_tal.ali` och `summera_tal.o` som innehåller temporär information som endast behövs under kompileringen eller för felsökning. Man startar programmet genom att helt enkelt skriva `summera_tal`.

```
zaza3 <13> summera_tal
Ange antal tal: 4
Mata in tal: 178
Mata in tal: 35
Mata in tal: 29
Mata in tal: 471
Summan blev:          713
zaza3 <14>
```

[1] Du kan läsa mer om installationen av GNAT på IDA på adressen http://www.ida.liu.se/education/ugrad/progkon/Ada

4. Övningar

I följande övningar är tanken att ni själva ska konstruera och testa enkla program i Ada. Till en början utgår ni från exempel som modifieras något, för att så småningen skriva hela program själva från grunden. Om ni inte har någon erfarenhet av programmering sedan tidigare är det rimligt att hinna med de fyra första övningarna under laborationspasset. De resterande övningarna får betraktas som en bonus och har markerats med asterisk.

En mycket kompakt sammanfattning av de viktigaste delarna av Ada finns på http://www.ida.liu.se/~petda/ada-summary.html

Om ni gör fel i programmet kommer kompilatorn att protestera med ett felmeddelande. Dessa meddelanden innehåller en referens som talar om på vilken rad och i vilken kolumn felet är. Om ni kör fast och inte lyckas lösa problemet, kontakta handledare.

Övning 1 – Komma igång

Kopiera exempelfilen från kurskontot, installera kompilatorn och pröva att kompilera och köra programmet enligt ovan! Öppna gärna exempelfilen i Emacs så att du kan titta på den. Läs igenom förklaringen av vad programmet gör och försök förstå ungefär hur det fungerar. Exempelfilen finns här:

```
~TDDC66/pp/summera_tal.adb
```

Kom ihåg att när du är klar med övningarna kan det vara lämpligt att ta bort skräpfilerna och framför allt de körbara programmen eftersom de senare brukar bli ganska stora. Spara endast dina källkodsfiler med efternamnet `.adb`.

Övning 2 – Mindre ändringar i programmet

Utskriften av slutsumman i exempelprogrammet ser lite konstig ut. Det beror på att utskriften av tal i Ada av någon outgrundlig anledning är förinställd på att alltid använda elva positioner för talen och högerjustera dem. I exemplet ovan skrivs det alltså ut nio mellanslag följt av de tre sifforna i talet 713. Man kan dock styra hur många positioner ett tal ska skrivas ut med så här:

```
Put(Summa, Width=>3);
```

Detta skulle göra att talet bara skrivs ut med tre positioner. Problemet är ju att vi inte vet hur stor summan är från början. Då kan vi ange bredden 0, för då kommer Ada att använda precis så många positioner som behövs.

Öppna filen i Emacs och ändra i exempelprogrammet så att summan skrivs ut snyggt enligt ovan, oavsett vad den blir. Ändra också inne i loopen så att programmet beter sig så här:

```
zaza3 <14> summera_tal
Ange antal tal: 3
Mata in tal 1: 14
Mata in tal 2: 15
Mata in tal 3: 16
Summan blev: 45
zaza3 <15>
```

Tips: Krångla inte till det genom att försöka skriva ut flera saker på samma rad. Antingen kan du skriva ut text eller tal, inte båda (åtminstone inte på något lätt sätt).

Övning 3 – Ett nytt program

Skriv ett nytt program som läser in ett tal och skriver ut multiplikationstabellen för det talet (som i exemplet nedan).

```
zaza3 <15> tabell
Ange vilken tabell du vill se: 7

Multiplikationstabell
---------------------
1 * 7 = 7
2 * 7 = 14
3 * 7 = 21
4 * 7 = 28
5 * 7 = 35
6 * 7 = 42
7 * 7 = 49
8 * 7 = 56
9 * 7 = 63
10 * 7 = 70
zaza3 <16>
```

Tips: Gör en kopia av exempelprogrammet Summera_Tal och utgå från den filen.

Övning 4

Skriv ett nytt program som läser in ett tal, beräknar fakulteten för det talet och skriver ut resultatet på skärmen. Fakulteten av n är som bekant produkten av alla tal från 1 till och med n.

Övning 5*

Skriv ett program som läser in ett tal n och beräknar det n:te Fibonacci-talet. Fibonacci-serien är 1, 1, 2, 3, 5, 8, 13, … dvs varje tal är summan av de två närmast föregående (förutom de två första som är 1).

Övning 6*

Modifiera programmet från övning 1 så att det förutom att skriva ut summan av alla talen också skriver ut vilket av talen som var minst. För att kunna göra detta behöver du en extra variabel som håller reda på vilket av talen som är det minsta hittills. Studera den Ada-sammanfattning på webben som refererades ovan, framför allt hur man gör jämförelser med `if` i Ada.

Övning 7*

Studera Ada-sammanfattningen och ta reda på hur man kan göra *underprogram*, dvs egna procedurer och funktioner. Använd sedan den kunskapen för att bygga upp ett program bestående av en meny där man kan välja att anropa procedurerna och funktionerna från övning 3-6. När ett av programmen kört klart ska man komma tillbaka till menyn. Avslutar gör man genom att välja det alternativet i menyn:

```
zaza3 <15> menyprogram
Vad vill du göra?
1. Skriva ut en multiplikationstabell
2. Beräkna en fakultet
3. Räkna ut ett Fibonacci-tal
4. Summera ett antal tal och skriv ut det minsta
5. Avsluta
Välj:
```

Tips: Använd `case`

5. Referenser

Framtida kurser

Programspråket Ada återkommer bland annat i kursen *Imperativ programmering och Ada* som C och D läser på våren i årskurs 1.

Litteratur

I den rekommenderade referensboken *Computer Science: An Overview* av J. Glenn Brookshear är särskilt avsnitt 6.1 och 6.2 intressanta för denna laboration.

Du kan läsa mer om Ada i någon av de böcker som används i vårens Ada-kurs. Som huvudalternativ finns *Programmering i Ada95* av Torbjörn Jonsson, som också är examinator i Ada-kurserna. Andra böcker som tidigare använts är *Ada från början* av Jan Skansholm och *Programming in Ada95* av J.G.P. Barnes. Den engelska boken är mer fullständig än de svenska, men är inte fullt lika pedagogiskt upplagd.

Du kan också läsa om Ada i *Concepts of Programming Languages* av Robert W. Sebesta. Det är en bok som tar upp många olika programspråk och sätt att programmera.

En mycket kompakt sammanfattning av den begränsade del av Ada som tas upp i vårens kurser finns på http://www.ida.liu.se/~petda/ada-summary.html

6. Frågor att fundera över

För dig som är nybörjare på programmering

Programmering är en process som består av flera faser. Först måste man förstå vad det är för problem som man ska lösa. Därefter ska man försöka designa en lösning. Denna lösning ska *implementeras*, dvs man ska skriva själva programmet. Sist men inte minst måste det färdiga programmet testas. Oftast går dock inte processen så här rakt och tydligt. Många gånger kan man tjäna på att experimentera lite, utan att ha förstått själva problemet. En del hävdar till och med att det är först när man har utformat lösningen som man verkligen har förstått problemet. För att bli en bra programmerare krävs lång träning – mycket längre än vad ens en högskoleutbildning kan ge. Man måste lära känna de olika byggstenarna som finns i programspråket, men också lära sig när och hur man ska använda dem.

Hur gjorde du när du löste övningarna i den här laborationen? Kan du känna igen de steg som beskrivs ovan? Känner du att du har förstått åtminstone lite grann av vilka byggstenar som finns i Ada? Kändes det svårt att försöka formulera lösningar på problem i ett formellt programmeringsspråk?

För dig som programmerat en del förut

Liknar Ada något annat språk som du har erfarenhet av? Vad känns svårare eller lättare att göra i Ada, enligt din bedömning?