UPPSALA
UNIVERSITET

Programmeringsmetodik DV1
Programkonstruktion 1

Ett programutvecklingsexempel

Produktionsplanering

Vid produktionsplanering i en fabrik använder man *komponentlistor* för att beskriva materialet som behövs för att tillverka en produkt. Komponentlistan anger de komponenter som behövs tillsammans med antalet av varje komponent.

En komponent är antingen en färdig *grundkomponent* som inte behöver tillverkas eller en *sammansatt komponent* som tillverkas på fabriken (och därmed själv är en produkt som har en komponentlista). En komponentlista kan innehålla både grundkomponenter och sammansatta komponenter. Komponentlistorna lagras i ett *produktregister*.

Skriv ett program som talar om vilka grundkomponenter som krävs för att tillverka en viss produkt och hur många av varje som behövs!

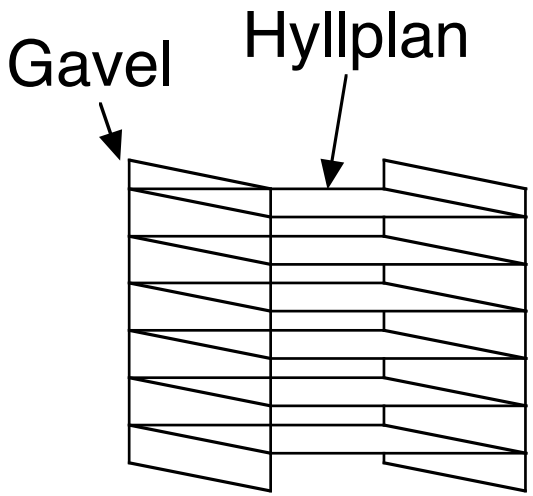
Exempel – produktregister

Sammansatta komponenter:

- bokhylla: 1 gavel, 1 hylldel
- bred bokhylla: 1 gavel, 2 hylldelar
- hylldel: 1 gavel, 6 hyllplan
- hörnbokhylla: 1 bokhylla, 1 hylldel, 1 hörnstöd

Grundkomponenter:

- gavel
- hyllplan
- hörnstöd



En hörnbokhylla består t.ex. av 3 gavlar, 12 hyllplan och 1 hörnstöd.

Programutvecklingsprocessen

1. Program(krav-)specifikation
2. Programdesign/problemlösning
3. Kodning
4. Kodgranskning
5. Testning
6. Felsökning
7. Dokumentation (görs parallellt med ovanstående aktiviteter)
8. Underhåll (arbete på programmet efter leverans)

Se minneslistan för programutveckling!

(Kurskompendiet eller
http://www.it.uu.se/edu/course/homepage/pkpm/HT06/minneslista)

1. Programspecifikation

1.1 Se till att du vet och förstår vad programmet skall göra.

1.1.1 Lägg speciell vikt vid att du förstår vad som skall hända vid "otypiska fall" som tomma datamängder.

1.1.2 Ställ frågor till beställaren om det behövs.

Hur identifierar man en produkt?

1.1.3 Om det är något som du inte kan ta reda på, gör ett rimligt antagande och dokumentera det!

Jag antar att alla komponenter finns i produktregistret med en enda komponentlista vardera.

Vidare att en produkt inte använder sig själv som komponent – varken direkt eller indirekt.

1.2 Skriv ned vad programmet skall göra som inte redan är skrivet.

Produkter identifieras med teckensträngar.

2. Programdesign

2.1 Tänk igenom så att du själv förstår hur problemet skall lösas.

2.1.1 Har någon annan redan löst problemet? Om det t.ex. finns en inbyggd funktion i ML som gör det som behövs så behöver du inte göra något själv!

Nej, det har knappast någon gjort i detta fall...

2.2 Konstruera datastrukturer.

2.3 Konstruera algoritmer

2.4 Skriv funktionsspecifikationer.

2.5 Skriv testfall.

2.2 Konstruera datastrukturer

2.2.1 Bestäm en lämplig representation för de data som programmet skall hantera…

En produktidentifikation representeras som en `string`*.*

En komponentlista representeras som en `(string*int) list` *där varje element är ett par (identifiering,antal) för en komponent.*
Vi kallar denna typ `partsList`*.*

Produktregistret representeras som en `(string*partsList) list` *där varje element är ett par (identifiering,komponentlista) för en sammansatt komponent.*
En grundkomponent finns i registret men har en tom komponentlista.

2.2 Konstruera datastrukturer (forts.)

2.2.2 …försök undvika att samma information kan representeras på olika sätt. Ett sätt är att bestämma en datastrukturinvariant som utesluter alla representationer utom en.

Ordningen av komponenter i en komponentlista eller produktregister är inte bestämd, men det accepterar vi.

Om en komponent inte används i en komponentlista så saknas den i listan ***eller*** *så kan antalet av komponenten vara noll. Vi bestämmer att den skall saknas.*

```
[("gavel",2),("hyllplan",6),("hörnstöd",0)]
```

är alltså otillåten – skall vara

```
[("gavel",2),("hyllplan",6)]
```

2.2 Konstruera datastrukturer (forts.)

2.2.3 Om datastrukturer kan ha instanser som inte representerar någon information alls (…), bestäm en datastrukturinvariant för att utesluta dessa fall.

Om en komponent förekommer flera gånger i en komponent- eller tabellista blir informationen tvetydig. En komponent får bara förekomma högst en gång i varje komponentlista.

Antalet angivna exemplar av en komponent i en komponentlista får inte vara negativt.

I ett produktregister får inte en produkt använda sig själv som komponent – varken direkt eller indirekt.

I ett produktregister måste alla komponenter till en produkt själva finnas med i produktregistret.

2.2 Konstruera datastrukturer (forts).

2.2.4 Skriv en beskrivning av hur information representeras av dina datastrukturer och vilka datastrukturinvarianter som finns.

Beskrivningen består av informationen ovan (fast organiserad på ett strukturerat sätt för varje datastruktur och inte efter punkter i minneslistan), samt exempel:

Bokhylleproduktregistret representeras som:

```
val exempelregister =
[("bokhylla",[("gavel",1),("hylldel",1)]),
 ("hylldel",[("gavel",1),("hyllplan",6)]),
 ("bred bokhylla",[("gavel",1),("hylldel",2)]),
 ("hörnbokhylla",
   [("bokhylla",1),("hylldel",1),("hörnstöd",1)]),
 ("gavel",[]), ("hyllplan",[]), ("hörnstöd",[])]
```

2.3 Konstruera algoritmer

2.3.1 Använd flödesschemor, dataflödesdiagram, pseudokod eller andra hjälpmedel för att konkretisera hur programmet skall fungera.

2.3.2 Dela upp problemet i delproblem som du löser separat (stepwise refinement). Gå igenom steg 2 igen för varje deluppgift.

*Delproblem är skrivna med **fetstil**.*

2.3.2.1 Det är ok att göra övriga steg … innan … delproblemen….

*1) **Leta upp** komponentlistan för komponenten i produktregistret.*

2) Är det en grundkomponent? Då fall behövs ett exemplar av komponenten själv. Klart!

*3) Annars, **beräkna sammanlagda antalet grundkomponenter för alla delkomponenterna**.*

2.4 Skriv funktionsspecifikationer

2.4.1 Bestäm funktionernas namn och typ.

2.4.2 Bestäm namn på funktionsargumenten…

2.4.3 Bestäm för- och eftervillkor

2.4.4 Ta med något typiskt anrop och resultat som körexempel.

```
partBreakDown(pid,preg)
TYPE: string*(string*partsList) list -> partsList
PRE: pid betecknar en komponent som finns i
     produktregistret preg.
POST: en förteckning av de grundkomponenter
      och det antal av varje som krävs för att
      bygga maskindelen pid i enlighet med
      informationen i produktregistret preg.
EXAMPLE:
 partBreakDown("hörnbokhylla",exempelregister)
 = [("hörnstöd",1),("hyllplan",12),("gavel",3)]
```

2.5 Skriv testfall

2.5.1 Exemplen från funk.spec. skall ingå bland testfallen.

Se ovan.

2.5.2 Ifall kravspecifikationen innehåller exempel så skall de ingå bland testfallen.

Finns inte här.

2.5.3 Testfallen skall täcka både typiska och otypiska användningar av programmet. Tänk speciellt på gränsfall!

```
partBreakDown("hyllplan",exempelregister)
= [("hyllplan",1)]
```

2.5.4 Ifall kravspecifikationen ger möjlighet för programmet att ge alternativa svar så kan du försöka skriva testfallen så att de tar hänsyn till det, eller så skissar du bara resultatet och fyller i detaljerna i steg 5 när du vet exakt hur programmet arbetar.

Gå igenom steg 2 igen för varje deluppgift…

*”**Leta upp** komponenten i produktregistret”*

2.1 Tänk igenom så att du själv förstår hur problemet skall lösas.

Detta är tydligen ett tabellsökningsproblem

2.1.1 Har någon annan redan löst problemet? …

Vi kan återanvända tabellsökningsprogrammet från kursmoment 6!

```
lookup(key,table)
Type: ''a*(''a*'b) list -> 'b
Pre:Exakt en post i tabellen table har nyckeln key
Post: table är en lista av par (k,v) där k är en
  nyckel och v ett tabellvärde. Värdet av lookup
  är det tabellvärde som motsvarar nyckeln key.
```

Komponentidentifierare har typ `string` och produktregister har typ `string*partsList = string*(string*int) list`, vilket passar typen hos `lookup`.

Beräkna sammanlagda antalet grund-komponenter för alla delkomponenterna

2.1 Tänk igenom så att du själv förstår hur problemet skall lösas.

2.2 Konstruera datastrukturer.

Inga nya datastrukturer behövs.

2.3 Konstruera algoritmer

1) Gå igenom steg 2 och 3 för varje given delkomponent.

*2) Tag en delkomponent och **beräkna dess grundkomponenter** (samma som huvudproblemet).*

*3) **Multiplicera antalet komponenter i delkomponentens grundkomponentlista** med antalet gånger delkomponenten behövs för att bygga produkten.*

*4) **Slå** successivt **ihop grundkomponentlistorna** från steg 3 till en. Klart.*

Beräkna sammanlagda antalet (forts.)

2.4 Skriv funktionsspecifikationer.

```
partsListBreakDown(pl,preg)
TYPE: partsList*(string*partsList) list
       -> partsList
PRE: pl är en korrekt komponentlista.
     preg är ett korrekt produktregister.
     Maskindelarna i pl finns i
     produktregistret preg.
POST: en förteckning av de grundkomponenter och
     det antal av varje som krävs för att bygga
     samtliga komponenter i pl i enlighet med
     informationen i produktregistret preg.
EXEMPEL:
 partsListBreakDown
   ([("gavel",1),("hylldel",2)],exempelregister) =
        [("hyllplan", 12), ("gavel", 3)]
```

Beräkna sammanlagda antalet (forts.)

2.5 Skriv testfall.

Exemplet ovan samt gränsfall:

```
partsListBreakDown([],exempelregister) = []
```

Fortsätt på samma sätt...

Multiplicera antal komponenter i en komponentlista med ett tal

1) Upprepa steg 2 för varje komponent i listan

2) Tag multiplicera för en komponent med det givna talet

3) Bilda ny lista av komponenter och resultaten från steg 2.

```
multPartsList(k,pl)
TYPE: int*partsList->partsList
PRE:  k>0. pl är en korrekt komponentlista.
POST: komponentlistan pl där alla kvantiteter
      multiplicerats med k
EXAMPLE: multPartsList(2,[("hyllplan",6),
                          ("gavel",1)]) =
           [("hyllplan",12),("gavel",2)]
```

Testa gränsfall:

```
multPartsList(2,[]) = []
```

Fortsätt på samma sätt...

Slå ihop två komponentlistor

1) Upprepa steg 2 för varje komponent i den ena listan

*2) Tag en komponent i den ena listan och **lägg till den komponenten till den andra listan**.*

```
mergePartsList(pl,pl')
TYPE: partsList*partsList -> partsList
PRE:  pl och pl' är korrekta komponentlistor.
POST: De två komponentlistorna pl och pl'
      sammanslagna.
EXAMPLE: mergePartsList([("hyllplan",6),
                         ("gavel",1)],
                        [("gavel",2)]) =
           [("gavel",3),("hyllplan",6)]
```

Testa gränsfall – ettdera argumentet tomt, andra inte tomt.

Fortsätt på samma sätt...

Lägg till ett antal exemplar av en komponent till en komponentlista

1) Sök efter komponenten i listan.

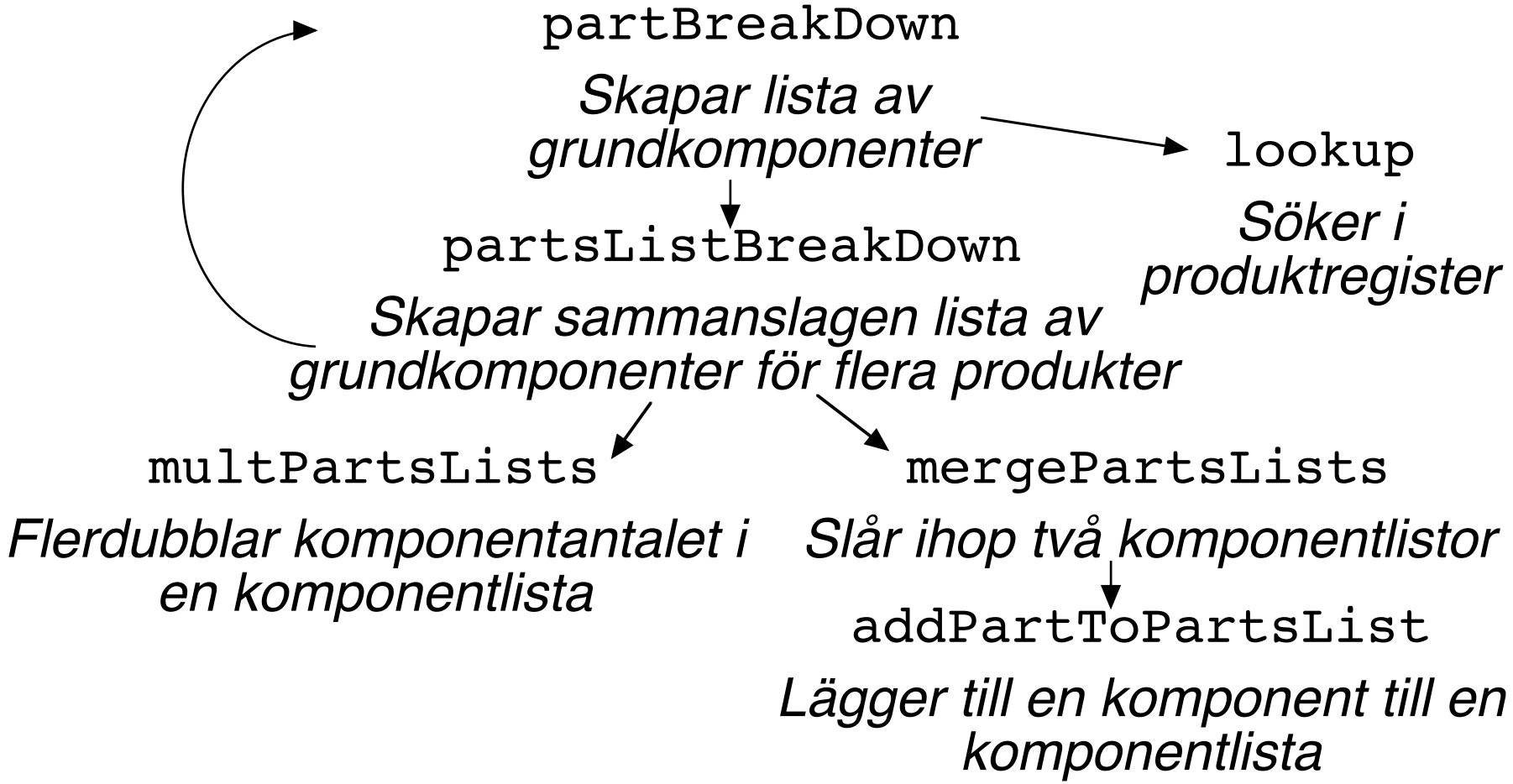
2) Finns den, så öka givet antal exemplar

3) Finns den inte, så lägg till den till listan.

```
addPartToPartsList(pid,n,pl)
TYPE: string*int*partsList -> partsList
PRE:  n>0. pl är en korrekt komponentlista.
POST: Komponentlistan pl med den nya
      komponenten pid tillagd i n exemplar.
EXAMPLE: addPartToPartsList("gavel",1,
                  [("hyllplan",6),("gavel",2)]) =
           [("gavel",3),("hyllplan",6)]
```

Mera testfall: tom lista, listan inte tom men komponenten finns inte i den…

Översikt över programstrukturen



3. Kodning (`partBreakDown`)

3.1 Använd flödesschemor, dataflödesdiagram eller andra hjälpmedel för att konkretisera hur funktionen skall fungera.

3.2 Behövs rekursion?

Inte direkt (men se kodningen av `partsListBreakDown`*).*

3.3 Bestäm de fall av indata som behöver hanteras på olika sätt…

Ingen skillnad.

3.4 Om man behöver använda olika delar av en datastruktur som beräknas av en uttryck…

Inte aktuellt.

```
fun partBreakDown(pid,preg) =
    case lookup(pid,preg) of
      [] => [(pid,1)]
    | pl => partsListBreakDown(pl,preg)
```

3. Kodning (`partsListBreakDown`)

3.2 Behövs rekursion? I så fall

Ja, för att gå igenom komponenterna i listan. Ett delproblem samma som huvudproblemet vilket i sig innebär rekursion.

3.2.1 Bestäm hur funktionen kan beräknas med hjälp av värdet för "enklare" indata. Använd flödesschemor, rekursiva ekvationer eller kända mönster. Svansrekursion med ackumulatorvariabel eller vanlig rekursion? Välj argumentet (argumenten) att göra rekursion över.

För att gå igenom komponenterna: standardmönstret för rekursion över listor.

För att anropa huvudfunktionen blir indata enklare eftersom de kräver färre antal uppslagningar av komponenter i produktregistret för att komma till grundkomponenterna.

huvudfunktionen.

3. Kodning (`partsListBreakDown`) forts.

3.2.2 Bestäm rekursionsvarianten, samt det värde den minskar mot, och dokumentera dessa.

Summan av längden på komponentlistan `pl` *och största antalet komponenter man successivt måste slå upp i registret för att komma till en grundkomponent. Den minskar mot 0 (tom lista).*

3.2.3 Tänk alltid på möjligheten att använda alternativ till rekursion som funktionerna `map`, `foldr` etc.

Det diskuteras i kursmoment 9.

3.3 Bestäm de fall av indata som behöver hanteras på olika sätt och skriv kod för dem. (Använd matchning och/eller if/case-uttryck för falluppdelningen.)

Tom och icke-tom komponentlista.

3. Kodning (`partsListBreakDown`) forts.

3.3.1 Vid rekursion, se till att det finns kod för basfallet (eller basfallen) och att den koden inte gör rekursiva anrop…

3.3.2 Vid rekursion, när du skriver kod för det allmänna fallet (fallen), använd resultatet från rekursiva anrop (med enklare indata) för att beräkna funktionsvärdet.

3.3.2.1 Tänk på att "enklare indata" betyder ett lägre värde på varianten.

3.3.3 Om det behövs, skriv defensiv kod för de fall när argumenten inte uppfyller förvillkoret.

```
and partsListBreakDown([],_) = []
  | partsListBreakDown((pid,n)::pl,preg) =
      mergePartsList(multPartsList(n,
                                   partBreakDown(pid,preg)),
                     partsListBreakDown(pl,preg));
```

3. Kodning (multPartsList, mergePartsLists)

Dessa funktioner kodas enligt ”standardmallen” för listor:

```
fun f(…,[],…) = …
  | f(…,first::rest,…) = … f(…,rest,…) …


(* VARIANT: längden av pl *)
fun multPartslist(k,[]) = []
  | multPartsList(k,(pid,n)::pl) =
      (pid,k*n)::multPartsList(k,pl);

(* VARIANT: längden av pl *)
fun mergePartsList([],pl') = pl'
  | mergePartsList((pid,n)::pl,pl') =
     mergePartsList(pl,
                    addPartToPartsList(pid,n,pl'));
```

3. Kodning (`addPartToPartsList`)

Lägg till ett antal exemplar av en komponent till en komponentlista

1) Sök efter komponenten i listan.

2) Finns den, så öka givet antal exemplar

3) Finns den inte, så lägg till den till listan.

Hur skall det göras?

3.2.1 Bestäm hur funktionen kan beräknas [rekursivt] med hjälp av värdet för "enklare" indata.

3. Kodning (`addPartToPartsList`) forts.

1) Listan tom? Då finns inte komponenten. Skapa lista med denna komponent. Klart!

2) Listan inte tom. Är den givna komponenten först i listan? Uppdatera i så fall antal exemplar. Klart!

3) Listan inte tom. Den givna komponenten är inte först i listan. Lägg till komponenten till resten av listan (rekursion). Lägg till den komponent som var först i listan.

```
(* VARIANT: längden av pl *)
fun addPartToPartsList(pid,n,[]) = [(pid,n)]
  | addPartToPartsList(pid,n,(pid',n')::pl) =
        if pid = pid' then
          (pid, n+n')::pl
        else
          (pid',n')::addPartToPartsList(pid,n,pl);
```

4. Kodgranskning

Granskingen av `mergePartsList` är exempel:

4.1 När du granskar koden får du (och måste) förutsätta att förvillkoren är uppfyllda och att ev. datastrukturinvarianter är uppfyllda för alla argument.

4.2 Läs igenom koden. Ser det vettigt ut? Förstår du själv det du har skrivit?

Funktionen lägger till första komponenten i `pl` *till* `pl'` *och lägger sedan till resten av* `pl`*.*

4.3 Kontrollera att det finns kod som hanterar alla olika värden som argumenten kan ha (och som är tänkbara enligt punkt 4.1).

Första argumentet kan vara tomt eller icke-tomt. Båda fall hanteras.

4.3.1 Denna koll måste du även göra för case-uttryck och liknande som finns inne i funktionen du granskar.

4. Kodgranskning (forts.)

4.4 Granska alla funktionsanrop och övertyga dig om att de anropade funktionernas förvillkor är uppfyllda (återigen tänk på 4.1).

- `pl` *är en korrekt komponentlista. Därför måste* `n>0.`
  `pl'` *är en korrekt komponentlista.*
  Förvillkoret till `addPartToPartsList` *är uppfyllt!*
- `pl` *är en korrekt komponentlista. Därför är* `rest` *också det.*
  `addPartToPartsList` *beräknar en korrekt komponentlista (enligt dess eftervillkor).*
  Förvillkoret till `mergePartsList` *är uppfyllt!*

4. Kodgranskning (forts.)

4.5 Med kännedom om eftervillkoret hos alla funktionsanrop, övertyga dig om att programmet beräknar ett värde som uppfyller eftervillkoret hos den funktion du granskar.

I detta fall, se resonemang under 4.2 ovan.

4.5.1 Använd inte programkoden för de anropade funktionerna, utan bara deras eftervillkor från funktionsspecifikationen.

4.6 Kontrollera att rekursionsvarianten minskar i … rekursiva anrop!

`pl`-*argumentet till* `mergePartsList` *är ett element kortare i det rekursiva anropet. Alltså minskar rekursionsvarianten.*

4.7 Kontrollera att alla datastrukturer som du konstruerar uppfyller sina invarianter (återigen tänk på 4.1)

I basfallet genom att `pl'` *enl. förvillkor är en korrekt komponentlista.*

I rek.fallet genom att funk. förutsätts göra rätt med kortare lista.

5. Testning

5.1 Kontrollera att de testfall du skrivit tillsammans gör att all kod i programmet utförs. Om inte, skriv kompletterande testfall.

Kör testfall för hand och stryk över utförd kod i en programutskrift på papper. Kontrollera att all kod utförts…

5.2 Skapa kod för automatisk testning. Om testfallen inte tar för lång tid att köra så kan du lägga testkoden i slutet av programfilen.

```
(1,multPartsList(2,[]) = []);
(2,multPartsList(2,[("hyllplan",6),("gavel",1)])=
     [("hyllplan",12),("gavel",2)]);
```

5.3 Lägg upp testningen så att du testar en funktion innan du testar de andra funktioner som använder den.

5.4 Se till att du använder aktuella versioner av funktioner när du testar. Ladda om programfiler vid behov. För att vara bergsäker, starta om ML.

6. Felsökning

6.1 Om du felsöker program med abstrakta datatyper så underlättar det ofta att (tillfälligt!) göra om de abstrakta datatyperna till vanliga datatyper.

Abstrakta datatyper behandlas i kursmoment 8.

6.2 Kontrollera först att testfallet är riktigt! Tänk på att kravspecifikationen ibland kan tillåta alternativa svar.

```
multPartsList(2,[("hyllplan",6),("gavel",1)]) ≠
   ≠ [("gavel",2),("hyllplan",12)])
```

6.3 Granska den felande funktionen igen med tanke på just de indata som användes i det misslyckade testfallet.

6.4 Förvissa dig om att ev. anropade funktioner gör rätt!

6. Felsökning (hypotetiskt fel)

6.5 Om inget annat fungerar, utför beräkningen i funktionen steg för steg för hand (eller interaktivt genom att mata in uttryck till ML) och se var ett felaktigt värde uppstår.

```
mergePartsList([("hyllplan",6),("gavel",1)],
               [("gavel",2)]) = []

mergePartsList([…],[("gavel",2)])
—> mergePartsList([("gavel",1)],
 addPartToPartsList("hyllplan",6,[("gavel",2)]))
—> mergePartsList([("gavel",1)],
 addPartToPartsList("hyllplan",6,[("gavel",2)]))
—> mergePartsList([("gavel",1)],
                  [("gavel",2),("hyllplan",6)])
—> mergePartsList([],
 addPartToPartsList("gavel",1,
                   [("gavel",2),("hyllplan",6)]))
—> []
```

6. Felsökning (hypotetiskt fel) forts.

Felet var alltså att `mergePartsList` *beräknade tom lista när första argumentet var tomt även om andra argumentet inte var tomt. Så här såg den felaktiga funktionen ut:*

```
fun mergePartsList([],pl') = []
  | mergePartsList((pid,n)::pl,pl') =
     mergePartsList(pl,
                    addPartToPartsList(pid,n,pl'));
```

Nu kan felet rättas....

```
fun mergePartsList([],pl') = pl'
  | mergePartsList((pid,n)::pl,pl') =
     mergePartsList(pl,
                    addPartToPartsList(pid,n,pl'));
```

7. Dokumentation

…görs parallellt med ovanstående aktiviteter.

De flesta punkter nedan har redan behandlats och upprepas inte.

7.1 Beskriv programmet ur användarsynpunkt

7.1.1 Vilken uppgift utför det?

7.1.1.1 Finns några begränsningar i hur det utför uppgiften?

7.1.1.2 Finns några kända fel eller brister?

7.1.2 Hur använder man det?

Programmet används genom att man anropar funktionen `partBreakDown`.

7.1.2.1 Hur skall indata se ut?

Första argumentet skall vara en sträng med produktidentifiering. Andra argumentet skall vara ett produktregister (enligt tidigare exempel).

7. Dokumentation (forts.)

7.1.2.2 Hur skall utdata tolkas?

Värdet är en komponentlista (enligt tidigare exempel) som beskriver de grundkomponenter som behövs och antalet av varje.

7.1.2.3 Hur är arbetsgången?

7.1.3 Hur installerar man det?

Programfilen `prodreg.sml` *läses in i ett ML-system.*

7.1.3.1 Har det några speciella krav på maskinen det körs på, annan programvara etc.

Ett SML-system skall vara installerat.

7.2 Beskriv prog. ur programmerarsynpunkt

7.2.1 Ge en översikt över hur programmet arbetar. Uppdelning i moduler (olika funktioner).

7.2.2 Beskriv alla datastrukturer

7.2.2.1 Beskriv hur information representeras av datastrukturerna

7.2.2.2 Beskriv datastrukturinvarianter

7.2.3 Beskriv algoritmerna som används

7.2.4 Beskriv varje funktion.

7.2.4.1 Informationen i funktionsspecifikationen skall ingå.

7.2.4.2 Förklara hur funktionen utför sin uppgift.

7.2.4.3 Speciellt svårförståelig kod bör förklaras i en kommentar på plats i programfilen.

Exempel på dokumentation: Se inlämningsuppgift 1!

Funktionen `partBreakDown`

```
(* partBreakDown(pid,preg)
   TYPE: string*(string*partsList) list->partsList
   PRE: pid är en maskindel som finns i
        produktregistret preg.
   POST: en förteckning av de grundkomponenter
         och det antal av varje som krävs för att
         bygga maskindelen pid i enlighet med
         informationen i produktregistret preg.
*)
(* VARIANT: Största antal antalet komponenter man
            successivt måste slå upp i registret
            för att komma till en grundkomponent.
*)
fun partBreakDown(pid,preg) =
   case lookup(pid,preg) of
     [] => [(pid,1)]
   | pl => partsListBreakDown(pl,preg)
```

Funktionen partsListBreakDown

```
(* partsListBreakDown(pl,preg)
   TYPE: partsList*(string*partsList) list
         -> partsList
   PRE: pl är en korrekt komponentlista.
        preg är ett korrekt produktregister.
        Maskindelarna i pl finns i
        produktregistret preg.
   POST: en förteckning av de grundkomponenter och
        det antal av varje som krävs för att bygga
         samtliga komponenter i pl i enlighet med
         informationen i produktregistret preg. *)
(* VARIANT: summan av längden av pl samt
            varianten för partBreakDown *)
and partsListBreakDown([],_) = []
  | partsListBreakDown((pid,n)::pl,preg) =
      mergePartsList(multPartsList(n,
                          partBreakDown(pid,preg)),
                     partsListBreakDown(pl,preg));
```

Funktionen `multPartsList`

```
(* multPartsList(k,pl)
   TYPE: int*partsList->partsList
   PRE:  k>0. pl är en korrekt komponentlista.
   POST: komponentlistan pl där alla kvantiteter
         multiplicerats med k
*)
(* VARIANT: längden av pl *)
fun multPartslist(k,[]) = []
  | multPartsList(k,(pid,n)::pl) =
      (pid,k*n)::multPartsList(k,pl);
```

Funktionen `mergePartsLists`

```
(* mergePartsList(pl,pl')
   TYPE: partsList*partsList -> partsList
   PRE:  pl och pl' är korrekta komponentlistor.
   POST: De två komponentlistorna pl och pl'
         sammanslagna.
*)
(* VARIANT: längden av pl *)
fun mergePartsList([],pl') = pl'
  | mergePartsList((pid,n)::pl,pl') =
     mergePartsList(pl,
                    addPartToPartsList(pid,n,pl'));
```

Funktionen addPartToPartsList

```
(* addPartToPartsList(pid,n,pl)
   TYPE: string*int*partsList -> partsList
   PRE:  n>0. pl är en korrekt komponentlista.
   POST: Komponentlistan pl med den nya
         komponenten pid tillagd i n exemplar.
*)
(* VARIANT: längden av pl *)
fun addPartToPartsList(pid,n,[]) = [(pid,n)]
  | addPartToPartsList(pid,n,(pid',n')::pl) =
        if pid = pid' then
          (pid, n+n')::pl
        else
          (pid',n')::addPartToPartsList(pid,n,pl);
```

Funktionen `lookup`

```
(* lookup(key,table)
   TYPE: ''a*(''a*'b) list -> 'b
   PRE: Exakt en post i tabellen table har nyckeln
        key
   POST: table är en lista av par (k,v) där k är
         en nyckel och v ett tabellvärde. Värdet
         av lookup är det tabellvärde som
         motsvarar nyckeln key,
   EXAMPLE: lookup("Mattias Wiggberg",
              [("Lars-Henrik Eriksson",1057),
               ("Mattias Wiggberg",3176)]) = 3176
*)
(* VARIANT: längden av table *)
fun lookup(key,(k,v)::rest) = if key = k then
                                v
                              else
                                lookup(key,rest)
```