UPPSALA
UNIVERSITET

Programmeringsmetodik DV1
Programkonstruktion 1

Moment 2
Introduktion till funktioner och programutveckling

Återanvändning av kod?

```
String.substring("Hägg",size "Hägg"-3,3)
—> "ägg"
```

Detta uttryck plockar ut ett visst antal tecken (3) ifrån slutet av en teckensträng (`"Hägg"`).

Vill man plocka ut ett annat antal tecken eller plocka ur en annan sträng så måste man skriva om uttrycket med delar utbytta:

```
String.substring("Ångström",size "Ångström"-5,5)
—> "ström"
```

Att behöva skriva om likadan kod tar tid och leder lätt till misstag (som att bara byta ut 3 mot 5 på ena stället).

Man vill göra koden oberoende av vilken sträng och teckenantal det gäller. Hur lösa detta problem?

Egna funktioner i ML

Man kan göra definiera *egna funktioner* (egentligen *funktions-procedurer*) med uttryck som

- är oberoende av speciella indata
- kan användas flera gånger utan kopiering
- ges indata varje gång de används.

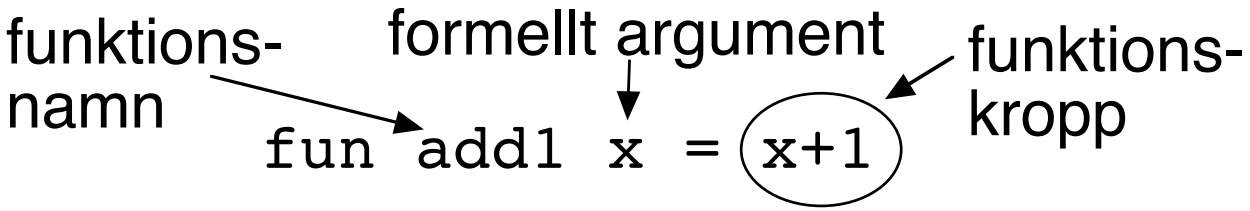
```
- fun last(s,n) =
      String.substring(s,size s-n,n);
> val last = fn : string * int -> string
```

Denna deklaration definierar en ny funktion `last` som tar fram tecken från slutet av en sträng!

```
- last("Hägg",3);
> val it = "ägg" : string
```

Ett ML-*program* byggs upp av sådana funktionsdefinitioner.

Funktionsdefinitioner



Namnet `add1` *binds* till det nya funktionsuttrycket av deklarationen.

När man *anropar* funktionen (*applicerar* den på argument) så *binds* `x` till argumentet i anropet (det *aktuella* argumentet) *medan* funktionskroppen beräknas. `x` kallas för en *bunden variabel*. Funktionskroppen beräknas *som om* `x` *bytts ut* mot det aktuella argumentet. Funktionen sägs *returnera* det beräknade värdet. Valet av `x` som identifierare är *godtyckligt*.

```
add1(4*2) ---> add1 8 ---> 8+1 ---> 9
```

Funktionens *typ* bestäms av typerna hos det formella argumentet och funktionskroppen. Funktionen ovan har typen `int -> int`.

Funktionsabstraktion

Att deklarera

```
fun last(s,n) = String.substring(s,size s-n,n);
```

kallas för en *funktionsabstraktion*.

Man har *abstraherat bort* konkreta data (t.ex. strängen `"Hägg"` och antalet 3) för att få ett mer generellt – abstrakt – uttryck.

Samtidigt utför man en *definitionsabstraktion* eftersom `last` blir ett namn för uttrycket.

Beräkningen av tecken i slutet av strängen `"Hägg"` har också blivit mer abstrakt genom att uttrycket

```
String.substring("Hägg",size "Hägg"-3,3)
```

ersatts av ett uttryck `last("Hägg",3)`, där den exakta metoden att ta fram de sista tecknen *inte framgår*.

Funktionsabstraktion i verkliga livet

Du vill skicka ett intyg till CSN-kontoret i Uppsala.

En metod är att be en kompis ta papperet och gå till CSN med det.

En annan är att stoppa det i ett kuvert och lägga i en brevlåda
Postens brevförmedling utför samma sak som din kompis men är…

- mer *abstrakt* eftersom du inte behöver veta hur det går till att få papperet till CSN.
- mer *generell* eftersom du kan använda posten för att skicka papper vart som helst (din kompis vill kanske inte åka till Sundsvall för att lämna ett papper till CSN huvudkontor).

Mer exempel....

```
fun add1 x = x+1;

add1 3+add1(4*2)
---> (3+1)+add1(4*2)
---> 4+add1(4*2)
---> 4+add1 8
---> 4+(8+1)
---> 4+9
---> 13

add1(3*add1(4*2))
---> add1(3*add1 8)
---> add1(3*(8+1))
---> add1(3*9)
---> add1 27
---> 27+1
---> 28
```

Ännu mer exempel...

```
fun last(s,n) = String.substring(s,size s-n,n);

last("Hägg",3)
--> String.substring("Hägg",size "Hägg"-3,3)
--> String.substring("Hägg",4-3,3)
--> String.substring("Hägg",1,3)
--> "ägg"

last("Ångström",5)
--> String.substring("Ångström",
                     size "Ångström"-5,5)
--> String.substring("Ångström",8-5,5)
--> String.substring("Ångström",3,5)
--> "ström"
```

Program på fil

Normalt skapar man sitt ML-program i en texteditor (t.ex. Emacs) och sparar på fil där namnet slutar med `.sml`.

`example.sml`

```
fun last(s,n) =
      String.substring(s,size s-n,n);
```

Filer *läses in* i ML med kommandot `use`. (Egentligen också en funktion – detta återkommer jag till i slutet av kursen.)

Filen behandlas (nästan) som om texten i filen skrivits in för hand.

```
- use "example.sml";
[opening file "example.sml"]
> val last = fn : string * int -> string
[closing file "example.sml"]
> val it = () : unit
- last("Hägg",3);
> val it = "ägg" : string
```

Inuti programfilen...

Programfiler innehåller typiskt en mängd deklarationer, men även uttryck kan finnas där.

Funktioner (och namngivna värden) måste definieras *innan* de används.

Använder man bibliotek som behöver laddas lägger man normalt ett `load`-uttryck i början av programfilen där biblioteket används.

Semikolon efter deklarationer och uttryck kan utelämnas om ML genom förekomsten av nyckelord kan avgöra var en deklaration eller ett uttryck slutar och nästa börjar.

Kommentarer `(* ... *)` läggs in i programfiler för att förklara programkoden där.

ML-mod i Emacs

Man kan köra ML inifrån Emacs. Detta är praktiskt – bl.a. kan man redigera och köra ML-program ifrån samma verktyg (Emacs).

Kommandot `M-x mosml` startar Moscow ML i en buffert med namnet `*mosml*`. Tryck return när Emacs frågar efter kommandonamn resp. argument.

I bufferten `*mosml*` skriver du uttryck, deklarationer etc. När du trycker på Return skickas texten på samma rad till ML.

Har du ett ML-program i en annan buffert kan du skicka programmet till ML med kommandot `C-c C-b` när du är i denna buffert.

Man kan ”bläddra” bland tidigare indata till ML med `M-p` och `M-n`.

Avbryta ML

Ett ML-program som kör kan avbrytas med `^C` (`C-c C-c` i Emacs, eller – om det inte fungerar – `C-q C-c` Return).

ML-systemet självt stoppas genom att anropa funktionen `quit()` eller med `^D` (`C-c C-d` i Emacs).

Programutvecklingsprocessen

- Program(krav-)specifikation
- Programdesign/problemlösning
- Kodning
- Kodgranskning
- Testning
- Felsökning
- Dokumentation (görs parallellt med ovanstående aktiviteter)
- Underhåll (arbete på programmet efter leverans)

Det är viktigt att vara systematisk för att få program av hög kvalitet!

Det finns många olika processmodeller för programutveckling. Denna liknar den klassiska *vattenfallsmodellen*. Mera om detta i kurserna Programmeringsmetodik 2 och Programvaruteknik.

Kalkylator för rationella tal

Kravspecifikation: Programmet skall kunna ta emot två rationella tal (bråktal) och ett av de fyra räknesätten, utföra detta och visa resultatet. Programmet skall heta `qcalc`.

Problemlösning (design) innebär att konstruera datastrukturer och algoritmer som utför den beskrivna uppgiften.

Huvudprincip: *stegvis förfining*.

Om problemet inte kan överblickas, bryt ned det i delproblem och tillämpa samma princip på dessa.

Överblickbara problem kan kodas direkt.

Rationella tal

Definition:

tal som kan skrivas som en kvot av två heltal (täljare och nämnare) där nämnaren ≠ 0. Mängden av rationella tal betecknas **Q**.

Räkneregler:

$$\frac{x1}{x2} + \frac{y1}{y2} = \frac{x1*y2+y1*x2}{x2*y2} \qquad \frac{x1}{x2} * \frac{y1}{y2} = \frac{x1*y1}{x2*y2}$$

$$\frac{x1}{x2} - \frac{y1}{y2} = \frac{x1*y2-y1*x2}{x2*y2} \qquad \frac{x1}{x2} / \frac{y1}{y2} = \frac{x1*y2}{x2*y1}$$

Tupler i ML

Ett rationellt tal består av *två* heltal som hör ihop.

Grupper av data kan hållas ihop i en *tupel* (post).

Tupler är *konstruerade* och inte grundläggande datatyper.

Syntax i ML: `(`data1`,`data2`,...,`data*n*`)` *n* är minst 2.

Motsvarande typ: typ1`*`typ2`*...*`typ*n*

Ex: `(1,6) : int*int`

Ex: `(1.0,"Hej",true) : real*string*bool`

Åtkomst till komponenter (fält): *#n* (där *n* är ett *heltal*) ger fält *n*.

Ex: `#2 (1.0,"Hej",true)` ⟶ `"Hej"`

(ML har också poster med namngivna fält.)

OBS! *# i sig* är ingen funktion – man kan inte skriva t.ex. `#(n+1)` för att få ut fält nummer n+1.

Datarepresentation

Ett rationellt tal representeras naturligt som ett talpar (2-tupel) av heltal, `int*int`, där första talet är täljaren och det andra nämnaren.

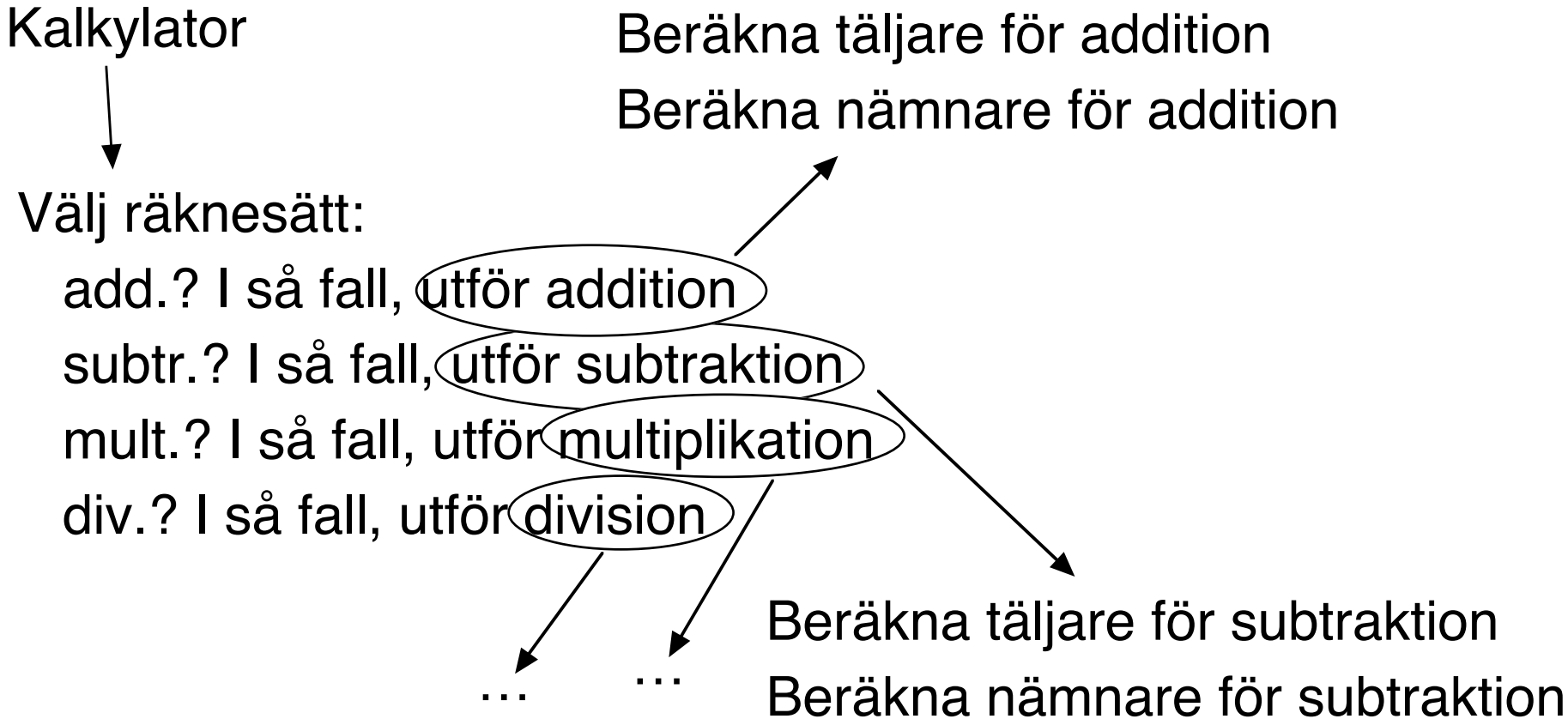
Alla 2-tupler av heltal representerar emellertid inte något rationellt tal – det gör inga tupler där andra talet (nämnaren) är 0.

Andra heltalet i varje par måste alltså vara skiljt från 0. Detta är en *invariant* – ett villkor som måste *bevaras* av programmet.

Räknesättet som skall utföras kan representeras av ett tecken, `#"+"`, `#"-"`, `#"*"` eller `#"/"`. Även här krävs en invariant – inget annat tecken får förekomma.

Andra representatoner är också tänkbara.

Problemlösning - stegvis förfining



Första förfiningssteget: skriv huvudfunktionen `qcalc` med typen

```
(int*int)*(int*int)*char -> int*int
```

Huvudfunktionen i kalkylatorprogrammet

```
fun qcalc(x:int*int,y:int*int,opr) =
  if opr = #"+" then
    qadd(x,y)
  else if opr = #"-" then
    qsub(x,y)
  else if opr = #"*" then
    qmul(x,y)
  else
    qdiv(x,y)
```

`x:int*int` betyder att man talar om för ML att `x` skall ha typen `int*int`. Mer om varför och när detta behövs kommer senare.

De ännu inte skrivna *underprogrammen* har kallats `qadd` etc.

Tack vare invarianten för räknesättet behöver man inte testa för `#"/"` – det är den enda återstående möjligheten.

Funktioner för de fyra räknesätten

```
fun qadd(x,y) =
  (#1 x * #2 y + #1 y * #2 x, #2 x * #2 y)
fun qsub(x,y) =
  (#1 x * #2 y - #1 y * #2 x, #2 x * #2 y)
fun qmul(x,y) =
  (#1 x * #1 y, #2 x * #2 y)
fun qdiv(x,y) =
  (#1 x * #2 y, #2 x * #1 y)
```

Dessa funktioner motsvarar nästa förfiningssteg.

De har alla typen `(int*int)*(int*int) -> int*int`

Vi har kodat `qcalc` först och underfunktionerna sedan (*top-down*). Alternativt kan man koda dessa underfunktionerna först och `qcalc` efteråt (*bottom-up*).

Exempel på räkning med rationella tal

```
qcalc((1,2),(1,3),#"+")
--> if #"+" = #"+" then qadd((1,2),(1,3)) else
if #"+" = #"-" then qsub((1,2),(1,3)) else if
#"+" = #"*" then qmul((1,2),(1,3)) else
qdiv((1,2),(1,3))
--> if true then qadd((1,2),(1,3)) else ……
--> qadd((1,2),(1,3))
--> (#1 (1,2) * #2 (1,3) + #1 (1,3) * #2 (1,2),
#2 (1,2) * #2 (1,3))
--> (1 * #2 (1,3) + #1 (1,3) * #2 (1,2), #2
(1,2) * #2 (1,3))
--> (1 * 3 + #1 (1,3) * #2 (1,2), #2 (1,2) * #2
(1,3))
--> (3 + #1 (1,3) * #2 (1,2), #2 (1,2) * #2
(1,3))
--> ……
--> (5,6)
```

Ett sammansatt exempel

Man kan förstås också använda funktionerna `qadd`, `qsub`, `qmul` och `qdiv` direkt för att utföra beräkningar:

Medelvärdet av 1/2 och 3/2:

```
qdiv(qplus((1,2),(3,2)),(2,1))
---> qdiv((#1(1,2)* #2(3,2)+ #1(3,2)* #2(1,2),
           #2(1,2)* #2(3,2)),
          (2,1))
---> qdiv((8,4),(2,1))
---> (#1(8,4)* #2(2,1),#2(2,1)* #1(8,4))
---> (8,8)
```

...vilket är det samma som 1/1.

(håll ordning på parenteserna....)

Invarianten

För att programmet skall vara riktigt krävs att den *bevarar* invarianten för rationella tal.

Om både `x` och `y` uppfyller invarianten, dvs `#2 x` ≠ 0 och `#2 y` ≠ 0 så måste även resultatet göra det, dvs `#2(qcalc(x,y,opr))` ≠ 0.

Man ser att detta gäller om det gäller för alla underprogrammen. Andra komponenten som beräknas av underprogrammen är:

```
qadd: #2 x * #2 y      qsub: #2 x * #2 y
qmul: #2 x * #2 y      qdiv: #2 x * #1 y
```

Invarianten är uppenbarligen uppfylld för de fyra räknesätten *utom* vid division x/y om täljaren till y = 0. Detta är inte så konstigt eftersom division med noll inte är meningsfullt!

Definitionsabstraktion igen

...”visa på en speciell användning av ett uttryck”.

Om vi vill ändra tecknet som anger multiplikation till `#"."`…

- dels måste man ändra i programmet (funktionen `qcalc`).
- dels måste man ändra i alla platser som *anropar* `qcalc`!
- om vi glömmer det ena.....eller ändrar där det inte skall ändras.....

Lösning: *namnge* tecknen som anger olika räknesätt:

```
val addition = #"+"
val subtraktion = #"–"
val multiplikation = #"*"
val division = #"/"
```

Kalkylatorprogrammet med namngivna konstanter

```
fun qcalc(x,y,opr) =
  if opr = addition then
    qadd(x,y)
  else if opr = subtraktion then
    qsub(x,y)
  else if opr = multiplikation then
    qmul(x,y)
  else
    qdiv(x,y)
```

Anropsexempel:

```
qcalc((1,2),(1,3),multiplikation) ⟶ (1,6)
```

Bara *ett* ställe (vid namngivningen) behöver ändras för att byta (t.ex.) multiplikationstecknet till #"."

Dessutom blir programkoden mer läsbar!

Exempel på räkning med namngivna konstanter

```
qcalc((1,2),(1,3),addition)
—> qcalc((1,2),(1,3),#"+")
—> if #"+" = addition then qadd((1,2),(1,3))
else if #"+" = subtraktion then
qsub((1,2),(1,3)) else if #"+" =
multiplikation then qmul((1,2),(1,3)) else
qdiv((1,2),(1,3))
—> if #"+" = #"+" then qadd((1,2),(1,3)) else
if #"+" = #"-" then qsub((1,2),(1,3)) else if
#"+" = #"*" then qmul((1,2),(1,3)) else
qdiv((1,2),(1,3))
—> if true then qadd((1,2),(1,3)) else ……
—> qadd((1,2),(1,3))
```

etc…….

Datarepresentationen igen

Normalt är två saker lika om deras delar är lika – och tvärtom.

```
qcalc((1,2),(1,3),addition) ⟶ (5,6)
qcalc((2,3),(1,6),addition) ⟶ (15,18)
```

Nu är 5/6 = 15/18, men 5 ≠ 15 och 6 ≠ 18!

Två *olika* representationer kan representera *samma* konkreta data. Detta leder till svårigheter när data skall jämföras i program.

En lösning: *Förfina* datarepresentationen så att den blir unik – t.ex. genom att kräva att rationella tal är maximalt förkortade (och att nämnaren är > 0) – en *ny* invariant. (Programmen måste ändras!)

En annan lösning: Skriv speciella jämförelseprogram, t.ex.

```
fun qequal(x,y) =
  #1 x * #2 y = #2 x * #1 y
```

Exempel på jämförelse

```
qequal((5,6),(15,18))
—> #1 (5,6) * #2 (15,18) = #2 (5,6) * #1 (15,18)
—> 5 * #2 (15,18) = #2 (5,6) * #1 (15,18)
—> 5 * 18 = #2 (5,6) * #1 (15,18)
—> 90 = #2 (5,6) * #1 (15,18)
—> 90 = 6 * #1 (15,18)
—> 90 = 6 * 15
—> 90 = 90
—> true
```

Gungor och karuseller...

Lösningen med förfinad datastruktur kräver extra beräkningar (förkortning) vid räkning med rationella tal, men kräver ingenting extra vid jämförelser.

Lösningen med speciella jämförelseprogram kräver inget extra vid räkning med rationella tal, men kräver extra beräkningar vid jämförelser.

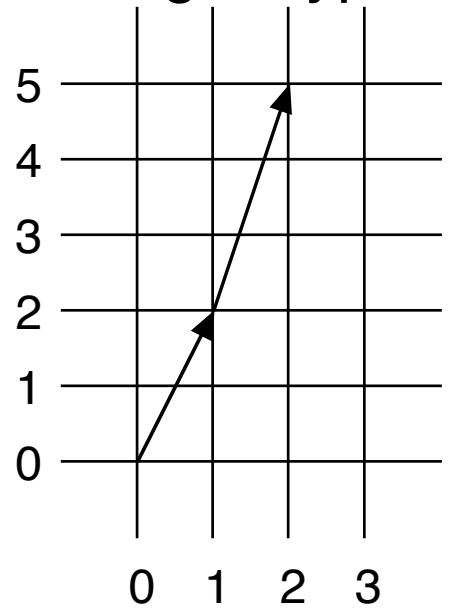
Extra beräkningar tar tid. Den bästa lösningen beror på proportionen mellan räkning och jämförelser.

(Men den ursprungliga representationen är dålig även av andra skäl – nämnaren växer hela tiden...overflow…)

Sammanblandning av representerade data

Rationella tal har speciella egenskaper och *borde* ha en egen typ.

Ex.vis. heltalsvektorer (punkter i rutnät eller matriser) representeras också naturligt som `int*int`-tupler – likadant som rationella tal.



Fastän datastrukturen är samma är egenskaperna helt annorlunda.T.ex. utförs addition av vektorer komponentvis – (1,2) + (1,3) = (2,5).

ML-program blir i praktiken *otypade* beträffande dessa sorters data.
Risk för sammanblanding: addition av vektorer med `qadd` eller blandning av rationella tal och vektorer.

Rationella tal och heltalsvektorer *borde* vara separata typer.
ML ger sådana möjligheter (som vi återkommer till senare...)

Kodgranskning

När programmet skrivits skall det *granskas* för att man skall försöka hitta felaktigheter.

Man läser koden och tänker igenom vad den gör och vad som är tveksamt. Det är viktigt att man helt och hållet *förstår* vad koden gör.

Kontrollera speciellt att
- Invarianter bevaras
- Förvillkor (kursmoment 3) är uppfyllda

I industriell programvaruutveckling görs granskningen av någon annan person och med hjälp av bestämda kodningsregler.

Testning

Vid programmering är risken stor att man gör misstag.
Syntaxfel, typfel o.dyl. upptäcks när programmet läses av datorn.
Fel på algoritmer (*logiska fel*) och beräkningsfel måste upptäckas med provkörning – *testning*.

Det finns väldigt (i princip oändligt) många sätt att använda ett program. Man måste testa med ett *urval* indata. Därför kan man aldrig med testning *garantera* att ett progam är felfritt. För det krävs matematisk analys av programmet – *formell verifiering*.

Grundläggande urvalsprinciper: man skall se till att testa…

- *alla* funktioner och *all* kod i varje funktion.
- *gränsfall* (inklusive *triviala* fall)
- *typiska* (icke-triviala) fall som täcker *hela kravspecifikationen*

Testfall

`qadd`: 2/3 + 1/4 = 11/12

`qsub`: 2/3 – 1/4 = 5/12

`qmul`: 2/3 * 1/4 = 1/6 (man måste ta hänsyn till att samma tal kan representeras olika....)

`qdiv`: 2/3 / 1/4 = 8/3

`qcalc`: Ett fall för var och en av hjälpfunktionerna (t.ex. fallen ovan).

Kontrollera att testfallen uppfyller provingskriterierna!

(I just detta program saknas egentliga gränsfall.)

Automatiserad testning

Så fort programmet ändras måste alla tester göras om!
Ett enkelt sätt att testa automatiskt är att lägga körningen av testfall i slutet av programfilen så körs de varje gång filen laddas.

Ett sätt att göra detta är att använda uttryck på formen

```
(n, testad kod = rätt värde);
```

n är ett nummer eller annan godtycklig identifiering av testfallet.
Detta uttryck beräknas till tupeln `(n,true)` om koden beräknade rätt värde och `(n,false)` annars.

Automatiserade testfall för `qcalc`

```
(1, qadd((2,3),(1,4)) = (11,12));
(2, qsub((2,3),(1,4)) = (5,12));
(3, qmul((2,3),(1,4)) = (2,12));
(4, qdiv((2,3),(1,4)) = (8,3));
(5, qcalc((2,3),(1,4),#"+") = (11,12));
```

……etc……

Utskrift när programfilen laddas:

```
- use "qcalc.sml";
[opening file "qcalc.sml"]
```

……utskrift från funktionsdefinitioner……

```
> val it = (1, true) : int * bool
> val it = (2, true) : int * bool
```

……etc……

```
[closing file "example.sml"]
> val it = () : unit
```

Felsökning

Eftersom tester aldrig kan vara heltäckande skall man alltid sträva efter att skriva programmet rätt från början – ingen ”trial and error”-programmering alltså! Oftast visar dock testerna att programmet ändå inte fungerar riktigt – då måste man *felsöka* programmet.

- Kontrollera först att testfallet verkligen är riktigt!
- Förvissa dig om att (ev.) hjälpfunktioner gör rätt (dessa bör vara färdigtestade och felsökta först).
- Granska den anropade funktionen igen med speciell tanke på de indata som användes i testfallet.
- Utför beräkninen steg för steg för att se var felaktiga värden uppstår. (Det förekommer felsökningshjälpmedel där datorn kan göra steg-för-stegberäkningar. Vi måste dock göra det för hand.)

Dokumentation

Beskrivning av programmet!

Dokumentation med olika syften:

- *Användardokumentation* – Hur skall man använda programmet?
- *Programdokumentation* – Hur är fungerar programmet?
  Beskrivning av algoritmer och datastrukturer
  Funktionen hos delprogram

Dokumentation på olika platser:

- *Intern* dokumentation (kodkommentarer i programfilen)
- *Extern* dokumentation (separata dokument)

Användardokumentation (exempel)

Programmet `qcalc` kan utföra de fyra räknesätten på två rationella tal (bråktal). `qcalc` tar tre argument: `x`, `y` och `opr`.

`opr` är ett tecken som beskriver det räknesätt som skall utföras: `#"+"` anger addition, `#"-"` anger subtraktion, `#"*"` anger multiplikation och `#"/"` anger division. Andra värden är inte tillåtna.

`x` och `y` är de två rationella tal som man skall räkna med. De representeras av heltalspar där första talet i paret är täljare och andra talet i paret är nämnare. Inga nämnare får vara 0. Vid division får inte det andra talets täljare vara 0.

Värdet av `qcalc` är det (oförkortade) resultatet som ett heltalspar.

Programdokumentation (exempel)

Datastrukturer: Rationella tal representeras med täljare och nämnare i en 2-tupel av typ `int*int`. Täljaren är första komponent och nämnaren andra komponent i tupeln. Nämnaren måste vara ≠0. Representationen är inte unik för ett givet rationellt tal. Räknesättet representeras med ett av tecknen `#"+"` (addition), `#"-"` (subtraktion), `#"*"` (multiplikation) eller `#"/"` (division).

Kodbeskrivning: `qcalc` är huvudfunktion och tar tre argument: `x`, `y` (rationella tal) och `opr` (räknesätt). Den väljer att anropa endera av `qadd`, `qsub`, `qmul` and `qdiv` baserat på vilken operation som angetts. Dessa funktioner tar två rationella tal som argument och utför direkt addition, subtraktion, multiplikation repektive division.

(Dessutom körexempel, för- och eftervillkor – se kursmoment 3.)

Dokumention av kursuppgifter

- Laborationerna:

  funktionstyper, för- och eftervillkor, varianter,
  sidoeffekter, körexempel
  (detta förklaras närmare i kursmoment 3).
  Lämnas som kodkommentarer.
  Se kodstandarden på kurswebben.

- Inlämningsuppgifterna:

  Komplett användardokumentation som separat dokument.
  Komplett programdokumentation uppdelat på separat
  dokument och kodkommentarer.
  Se dokumentationsexempel på kurswebben.

Varför bry sig?

... om systematisk programutveckling, välskriven kod, noggrann granskning och testning, god dokumentation etc.

- Så att programmet fungerar väl.
- Så att andra (t.ex. kursassistenterna...) förstår programmet nu.
- Så att *du* förstår programmet om ett halvår.
- Så att andra som skall ändra i ditt program förstår det *om 5 år*.

Grovt räknat utgör kostnader för *underhåll* – framtida ändringar av ett program – hälften av totalkostnaden för en programvara under dess livstid. Underhållet kan vara motiverat av upptäckta fel, ändrade förutsättningar eller nya önskemål om programfunktionen. Kan man mimimera underhållsinsatsen kan mycket pengar sparas!