UPPSALA
UNIVERSITET

Programmeringsmetodik DV1
Programkonstruktion 1

Moment 3
Mer om funktioner
och bindningar

Minns ni kalkylatorprogrammet?

```
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)

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)
```

Förvillkor

Funktioner är inte meningsfulla att anropa för alla argument.
T.ex. är det inte meningsfullt att anropa division om andra argumentet är 0: `1.0/0.0`, `1 div 0` eller `qdiv((1,2),(0,2))`.

Villkoren för att ett (del)program (funktion) är meningsfullt att köra kallas dess *förvillkor*.

För kalkylatorprogrammet gäller som förvillkor:

- De två ingående talens täljare och nämnare måste vara heltal.
  (I ML garanterar typsystemet att detta är uppfyllt, därför struntar man ofta i att ange det som uttryckligt förvillkor.)
- Inga nämnare får vara 0. (En datastrukturinvariant.)
- Vid division får det andra talets täljare inte vara 0.
- Räknesättet måste vara `#"+"`, `#"-"`, `#"*"` eller `#"/"` (också inv.)

Mer exempel på förvillkor

För `String.substring` gäller som förvillkor:

- Funktionen anropas med tre argument: en sträng (s),
  ett heltal (p) och ytterligare ett heltal (n).
  (I ML garanterar typsystemet detta).
- n ≥ 0 (Delsträngen kan inte ha negativ längd.)
- p ≥ 0 (Man kan inte ta ut delsträngen före huvudsträngens början.)
- p+n ≤ size s (Man kan inte ta ut delsträngen efter
  huvudsträngens slut.)

Funktionerna ^, + m.fl. *saknar* förvillkor (bortsett från typvillkoren)
– man kan alltid sätta ihop två strängar eller addera två tal!

(Egentligen är det inte sant eftersom det finns en maximal storlek för data, men i praktiken bortser man ofta från det.)

Om förvillkoret inte är uppfyllt...

Den europeiska Ariane 5-raketen havererar 40 sekunder efter uppskjutningen i juni 1996.

Programvara för Ariane 4 som återanvändes i Ariane 5-projektet kunde inte arbeta korrekt med den högre hastighet som Ariane 5-raketen hade. Man hade inte beaktat *förvillkoren* för programmet!

Observera att programmet *i sig* fungerade korrekt men i ett sammanhang det inte var avsett för.

Eftervillkor

En beskrivning av vad (del)programmet (funktionen) åstadkommer – dvs dess värde – är dess *eftervillkor*.

För `qcalc` gäller som eftervillkor: Värdet av `qcalc(x,y,opr)` är resultatet av att använda räknesättet beskrivet av `opr` på de rationella talen `x` och `y`.

För `String.substring` gäller som eftervillkor: Värdet av `String.substring(s,p,n)` är en sträng som omfattar `n` tecken i följd tagna från strängen `s` med början vid position `p`.

Användning av för- och eftervillkor

För- och eftervillkor kan användas i olika syften:

- För att beskriva vad ett program *skall* göra
  (i kravspecifikationen)
- För att beskriva vad ett program *faktiskt* gör
  (i programdokumentationen)

Vad programmet faktiskt gör och vad man kräver att det skall göra *kan* vara samma sak, men det är också möjligt att programmet både gör mer eller mindre (eller annorlunda) än vad man begärt. (Gör programmet mindre eller annorlunda är det förstås i någon mening felaktigt.)

”Design by contract”

För- och eftervillkor kan ses som ett kontrakt mellan programmet (funktionen) och den som använder (anropar) det.

Användaren lovar att uppfylla förvillkoren och programmet lovar att uppfylla eftervillkoren.

Detta betyder speciellt att om anroparen bryter sin del av avtalet (förvillkoret är inte är uppfyllt) så är programmet löst från kontraktet och får bete sig *hur som helst*.

Exempel: `qcalc((1,2),(1,0),#"/")` ⟶ `(0,2)`

Funktionsspecifikationer

Varje funktion i inlämnde uppgifter på kursen skall inledas av en *funktionsspecifikation* i form av en kodkommentar. Detta är en form av *intern programdokumentation*.

```
(* String.substring(s,p,n)
   Type: string*int*int -> string
   Pre:  0 <= p <= size(s), 0 <= n <= size(s)-p
   Post: Värdet är en delsträng av s med längd n
         och början på plats p i s.
   Ex.:  String.substring("abcd", 0, 3) = "abc"
         String.substring("abcd", 1, 2) = "bc"
         String.substring("abcd", 3, 1) = "d"
         String.substring("abcd", 4, 0) = ""
*)
```

Förkortningarna `Pre` och `Post` avser de engelska namnen på för- och eftervillkor: *pre*- resp. *postcondition*.

Funktionsspecifikation för `qcalc`

```
(* qcalc(x,y,opr)
   Type: (int*int)*(int*int)*char->(int*int)
   Pre:  x och y är rationella tal,
         opr är ett korrekt räknesätt,
         om opr = #"/" så får inte y vara 0.
   Post: Ett rationellt tal som är resultatet av
         räknesättet opr använt på x och y
   Ex.:  qcalc((1,2),(1,3),#"+") = (5,6)
*)
```

Att tänka på:

Förvillkoret beskriver vilka de acceptabla argumenten är för *din funktion*. Följer argumenten förvillkoret *måste* funktionen ge meningsfullt resultat, *får inte* ge exekveringsfel etc. Däremot *får* funktionen ge meningsfulla resultat även om förvillkoret inte är uppfyllt (men då är kanske förvillkoret onödigt starkt).

Eftervillkoret beskriver resultatet (värdet) av att anropa funktionen. Var tydlig – inga oklara syftningar! Om du inte använder varje argumentnamn i beskrivningen är ditt eftervillkor *nästan* säkert fel. *Varför*???

Defensiv programmering

Om förvillkoret inte är uppfyllt så kan ”vad som helst” hända.

Det är upp till programmeraren att förvissa sig om att programmen aldrig kommer att anropas utan att förvillkoren är uppfyllda.

I starkt typade språk som ML kontrollerar typmekanismen i viss utsträckning förvillkoren.

Programmet kan också *själv* kontrollera indata och vidta lämplig åtgärd om förvillkoret inte är uppfyllt (t.ex. avbryta beräkningen).

Detta kallas *defensiv* programmering.

Defensiv programmering av `qcalc`

```
fun qcalc(x:int*int,y:int*int,opr) =
  if #2 x = 0 orelse #2 y = 0 then
    raise NotRational
  else if opr = #"+" then
    qadd(x,y)
  else if opr = #"-" then
    qsub(x,y)
  else if opr = #"*" then
    qmul(x,y)
  else if opr = #"/" then
    qdiv(x,y)
  else
    raise WrongOperation
```

`raise` är en funktion som *avbryter* beräkningen med felkod.

Ändringar till `qdiv` beskrivs på nästa bild……

Defensiv programmering av `qcalc` (forts.)

```
exception NotRational
exception WrongOperation
fun qdiv(x,y) = if #1 y = 0 then
                 raise Div
                else
                 (#1 x * #2 y, #2 x * #1 y)
```

Felkoderna `NotRational` och `WrongOperation` måste deklareras.

Felkoden `Div` finns färdig i ML och används just för division med 0.

Var skall defensiv kod läggas?

- Varför inte göra division-inte-med-0 kontrollen i `qcalc`?
- Varför inte göra rationella-tal kontrollen i `qadd`,`qsub`,`qmul`,`qdiv`?

Defensiv kod tar utrymme (och beräkningstid) och man vill därför göra kontrollerna så tidigt som möjligt och lämpligt.

I en sammansatt beräkning *garanteras* det att argumenten till `qadd` etc. är korrekta rationella tal om dessa funktioner uppfyller datastrukturinvarianterna och ingångsvärdena är riktiga.

Eftersom ingångsvärdena går genom `qcalc` bör kontrollen av rationella tal läggas där. Det är svårare att garantera att andra argumentet till `qdiv` inte är noll och denna kontroll bör därför ligga där.

Automatisk testning av defensiv kod

Minns ni de automatiserade testfallen?

```
(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));
```
……etc……

På detta sätt kan man inte utan vidare testa fall som *skall* leda till att programkörningen avbryts. Testning av all kod i `qdiv`, t.ex., kräver att man kontrollerar att fallet 1/2 / 0/2 ger felkod `Div`.

Man kan då skriva testfall på formen:

`(n, (`testad kod`;false) handle` felkod => true`);`

T.ex.

```
(5, (qdiv((1,2),(0,2));false)handle Div=>true);
```

(Vad detta uttryck *egentligen* betyder återkommer vi till senare...)

Typhärledning

ML kan (nästan alltid) avgöra typen hos en funktion själv.

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

- `String.substring` har typen `string*int*int -> string`.
- Eftersom `s` är första argument till `String.substring` så måste `s` ha typen `string`.
- Eftersom `n` är sista argument till `String.substring` så måste `n` ha typen `int`.
- Därmed måste `last` ha argumenttypen `string*int`.
- Eftersom värdet av `String.substring` också är värdet av `last` så måste `last` ha värdetypen `string`.
- Alltså har `last` typen `string*int -> string`.

Detta kallas *typhärledning*.

Misslyckad typhärledning

Om ett program är feltypat så misslyckas typhärledningen.

```
- fun last(s,n) = String.substring(s,size s-n,s);
! Toplevel input:
! fun last(s,n) = String.substring(s,size s-n,s);
!                                              ^
! Type clash: expression of type
!   string
! cannot have type
!   int
```

Här har man av misstag gett `s` som sista argument till `String.substring`.

- `String.substring` har typen `string*int*int -> string`.
- `s` är 1:a argument till `String.substring` så typen är `string`.
- `s` är 3:e argument till `String.substring` så typen är `int`.

`s` kan inte vara `int` och `string` samtidigt – programmet är feltypat.

Oväntat resultat av typhärledning.

Vilken typ har funktionen `sumsquares`?

```
fun sumsquares(x,y) = x*x+y*y
```

`*` och + är båda överlagrade och har de alternativa typningarna `int*int->int` och `real*real->real`.

Om ML inte kan avgöra typen på aritmetiska operationer så väljer den `int`! `sumsquares` får alltså typen `int*int->int`.

Var avsikten att använda `sumsquares` på flyttal så får man typfel.
Lösning: ange uttrycklig typ på något deluttryck!

```
fun sumsquares(x:real,y) = x*x+y*y
```

eller

```
fun sumsquares(x,y) = (x*x+y*y):real
```

Tupler kan ge problem med typhärledning.

Varför var vi tvungna att deklarera argumenttyperna för `qcalc`?

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

Hjälpfunktionerna plockar ut 1:a och 2:a komponent ur tupler:

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

ML kan inte veta om tuplerna skall ha två, tre eller flera komponenter och därför inte bestämma deras typ.

Typhärledning går inte att genomföra.

Genom att på lämpligt ställe uttryckligen ange typen av indata till `int*int` försvinner problemet.

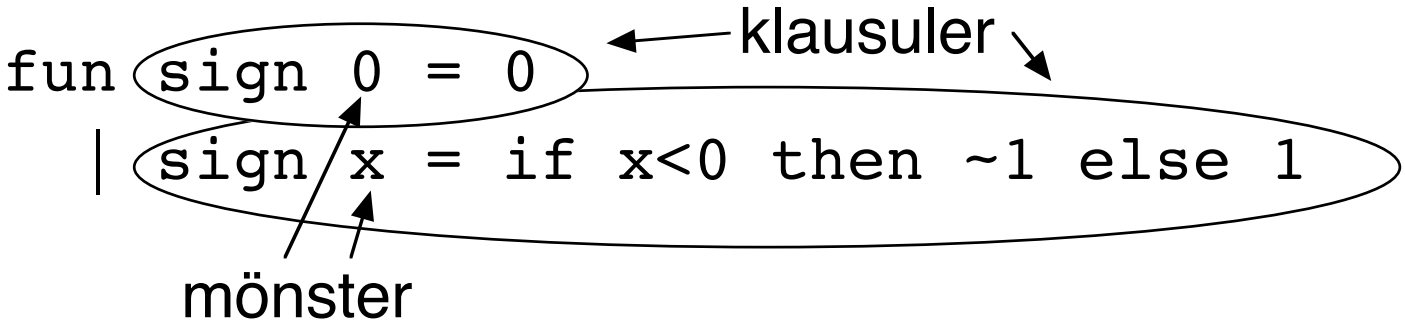
Mönstermatchning

De flesta funktioner skrivs med kod för flera olika *fall*.

Den grundläggande konstruktionen för falluppdelning är if-then-else.

```
fun sign x =
      if x = 0 then 0 else if x < 0 then ~1 else 1
```

Falluppdelning som jämför argument med *konstanter* kan göras direkt i funktionsuttrycket



När funktionen anropas går ML igenom klausulerna i tur och ordning och ser väljer den första klausul där argumenten ”passar”.

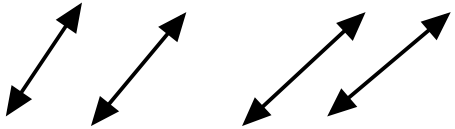
Matchning av sammansatta datatyper

Matchning kan också göras över *formen* på tupler.

Identifierare binds till *komponenterna* i tuplerna.

Tag t.ex. (ickedefensiva) `qdiv`. Denna funktion kan skrivas

```
fun qdiv((x1,x2),(y1,y2)) = (x1*y2,x2*y1)
```



```
qdiv((1,2),(3,4)) ---> (1*4,2*3) ---> (4,6)
```

`x1`, `x2`, `y1` och `y2` byts ut mot respektive `1`, `2`, `3` och `4`.

I praktiken använder men oftast (där det är möjligt) denna metod (snarare än #) för att komma åt komponenter i tupler eftersom det ger mycket mer lättläst kod.

Dessutom slipper man oklarheten kring antalet element i tuplerna – det blir helt klart att varje tupel innehåller 2 komponenter.

Funktioner med flera argument har bara ett

Funktioner i ML har *egentligen* alltid *ett* argument.
Om funktionen ser ut att ha flera argument, t.ex.

```
fun last(s,n) = ......
last("Hägg", 3) ⟶ "ägg"
```

...så konstrueras vid anropet egentligen en tupel av argumenten och skickas som *ett* argument till funktionen. Formen hos de formella argumenten i funktionsdefinitionen är egentligen bara mönstermatchning i funktionens enda riktiga argument.

Funktionstypen `string*int -> string` i detta exempel visar också det – argumentet är en `string*int`-tupel.
Av *praktiska* skäl säger man ofta ändå att `last` har två argument.

Matchning både på form och konstanter

Alternativ definition av `qdiv`. (Defensiv version.)

```
fun qdiv(_,(0,_))           = raise Div
  | qdiv((x1,x2),(y1,y2)) = (x1*y2,y1*x2)
```

_ (understrykningstecken) kan användas istället för identifierare när man inte är intreserad av argumentets värde.

Anropet	matchar
`qdiv((1,2),(0,4))`	första klausulen
`qdiv((1,2),(3,4))`	andra klausulen

Allmän form hos funktionsdefinitioner

```
fun namn mönster1 = uttryck1
  | namn mönster2 = uttryck2
    ......
  | namn mönsterN = uttryckN
```

Varje mönster är en konstant, en identifierare, symbolen _ eller ”skelettet” till en datastruktur (som tupler) där komponenterna i sin tur är mönster.

Alla mönster måste ha samma typ (funktionens argumenttyp).
Alla uttryck måste ha samma typ (funktionens värdetyp).

Se upp med matchning 1

Det är viktigt att klausulerna kommer i rätt ordning!

Alternativ definition av `qdiv` igen.

```
fun qdiv((x1,x2),(y1,y2)) = (x1*y2,y1*x2)
  | qdiv(_,(0,_))         = raise Div
```

```
qdiv((1,2),(0,4)) ⟶ (8,0)
```

Se upp med matchning 2

När man använder matchning mot datatyper där värdena är i form av identifierare (t.ex. `bool`) kan små felskrivningar ge syntaktiskt korrekta program med helt annorlunda beteenden!

```
fun boolName true = "Sant"
  | boolName false = "Falskt"
```

```
boolName true  ⟶ "Sant"
boolName false ⟶ "Falskt"
```

Låt oss göra en liten felskrivning:

```
fun boolName truw = "Sant"
  | boolName false = "Falskt"
```

```
boolName true  ⟶ "Sant"
boolName false ⟶ "Sant"
```

Varför??

Se upp med matchning 3

Vid ett funktionsanrop måste argumenten matcha *någon* klausul:

```
fun nameToBool "Sant" = true
  | nameToBool "Falskt" = false
```

`nameToBool "Sant"` —> `true`

`nameToBool "Falskt"` —> `false`

`nameToBool "Kanske"` —> exekveringsfel (`Match`)

I detta fall får man en varning när funktionen definieras.

```
- fun nameToBool "Sant" = true
    | nameToBool "Falskt" = false;
! Toplevel input:
! ....nameToBool "Sant" = true
!    | nameToBool "Falskt" = false.
! Warning: pattern matching is not exhaustive
> val nameToBool = fn : string -> bool
```

”Catchall”-klausuler

För att undvika exekveringsfel pga att ingen klausul matchar kan man lägga in en sista klausul som säkert matchar alla argument:

```
fun talNamn 1 = "Ett"
  | talNamn 2 = "Två"
  | talNamn 3 = "Tre"
  | talNamn _ = "Många"
```

Eller t.o.m.

```
fun talNamn 1 = "Ett"
  | talNamn 2 = "Två"
  | talNamn 3 = "Tre"
  | talNamn x = if x <= 0 then
                  "Lite"
                else
                  "Många"
```

Kalkylatorprogrammet med matchning

```
fun qadd((x1,x2),(y1,y2)) =
      (x1*y2 + y1*x2, x2*y2)

fun qsub((x1,x2),(y1,y2)) =
      (x1*y2 - y1*x2, x2*y2)

fun qmul((x1,x2),(y1,y2)) =
      (x1*y1, x2*y2)

fun qdiv((x1,x2),(y1,y2)) =
      (x1*y2, x2*y1)

fun qcalc(x,y,#"+") = qadd(x,y)
  | qcalc(x,y,#"-") = qsub(x,y)
  | qcalc(x,y,#"*") = qmul(x,y)
  | qcalc(x,y,_)    = qdiv(x,y)
```

Fördelar: Mycket mera läsligt.

Entydiga tupeltyper – typdeklarationer behövs inte.

Nackdel: Namngivna konstanter kan inte användas. (Varför?)

Defensivt kalkylatorprogram med matchning

Endast de funktioner som skiljer från förra bild…

```
exception NotRational

fun qdiv(_,(0,_))           = raise Div
  | qdiv((x1,x2),(y1,y2)) = (x1*y2, x2*y1)

fun qcalc((_,0),_,_) = raise NotRational
  | qcalc(_,(_,0),_) = raise NotRational
  | qcalc(x,y,#"+")  = qadd(x,y)
  | qcalc(x,y,#"-")  = qsub(x,y)
  | qcalc(x,y,#"*")  = qmul(x,y)
  | qcalc(x,y,#"/")  = qdiv(x,y)
```

Felaktigt räknesätt upptäcks genom att ingen klausul matchar.

Man kunde också ha lagt till en speciell klausul sist i `qcalc`, t.ex.

```
  | qcalc(_,_,_) = raise WrongOperation
```

Funktionsspecifikationer och matchning

```
(* qdiv(x,y)
   Type: (int*int)*(int*int)->(int*int)
   Pre: x och y är rationella tal. y är inte noll.
   Post: Kvoten av x och y
*)
    fun qdiv(_,(0,_))           = raise Div
      | qdiv((x1,x2),(y1,y2)) = (x1*y2,y1*x2)
```

Namnen `x` och `y` används här för att referera till argumenten *inom* funktions*specifikationen*. De har ingenting att göra med namnen på formella argument i funktions*deklarationen*! Det behöver inte ens vara samma namn – se på koden i exemplet.

Ibland ser man specifikationer som:

```
   Pre: (x1,x2) och (y1,y2) är rationella tal.
        (0,_) är inte noll.
```

Detta är *fel*!!!

Case-uttryck

Matchning väljer olika fall beroende på argumenten till en funktion. Matchning kan också användas för att välja olika fall beroende på värdet av ett uttryck.

```
fun dumMultiplikation(x,y) =
      case x*y of
        1 => "Ett"
      | 2 => "Två"
      | 3 => "Tre"
      | 4 => "Fyra"
      | _ => "Mycket"
```

```
dumMultiplikation(2,2) ⟶ "Fyra"
dumMultiplikation(2,3) ⟶ "Mycket"
```

Liksom i if-then-else så beräknas inte uttrycken i case-klausulerna förrän man vet vilket alternativ som valts.

Bindning i case-mönster

```
fun pratsamMultiplikation(x,y) =
      case x*y of
        1 => "Ett"
      | 2 => "Två"
      | 3 => "Tre"
      | 4 => "Fyra"
      | z => Int.toString z ^ "."
```

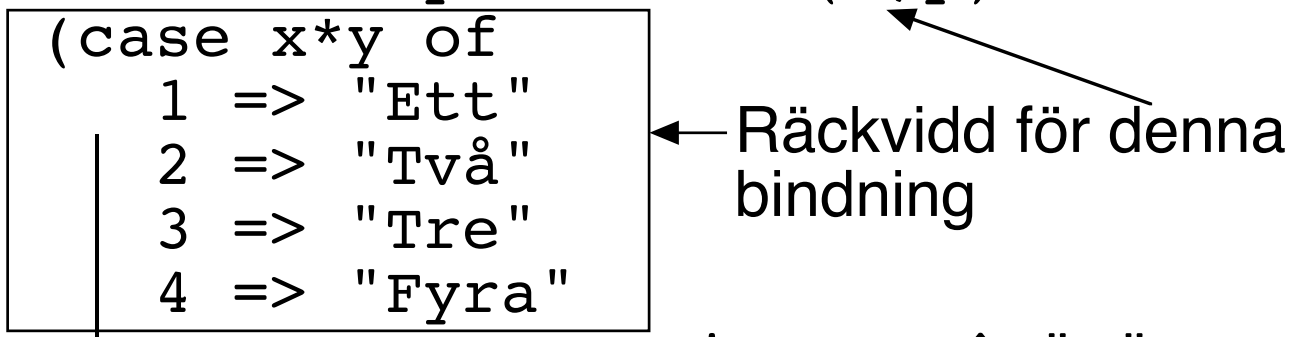
```
pratsamMultiplikation(2,3)
—> case 2*3 of 1 => "Ett" | 2 => "Två" | ……
—> case 6 of 1 => "Ett" | 2 => "Två" | ……
—> Int.toString 6 ^ "."
—> "6" ^ "."
—> "6."
```

Identifierare i mönstren byts alltså ut i klausulerna mot uttrycket efter case på liknande sätt som bundna identifierare byts mot argumenten i funktionsanrop.

Bindningsräckvidd

Vi har sagt att bundna variabler i funktionskroppen byts ut mot respektive aktuella argument när en funktion anropas.
I verkligheten är det lite mer komplicerat.

```
fun pratsamMultiplikation2(x,y) =
    (case x*y of
       1 => "Ett"
     | 2 => "Två"
     | 3 => "Tre"
     | 4 => "Fyra"
     | x => Int.toString x) ^ "."
```



Vid ett anrop `pratsamMultiplikation(2,3)` byts *inte* `x` ut mot `2` i den sista case-klausulen. Bindningen av `x` i case-uttrycket *skuggar* den första bindningen. *Räckvidden* för bindningen av `x` i funktionsdefinitionen innehåller inte sista case-klausulen.

Vad händer om variabler binds igen?

```
fun pratsamMultiplikation2(x,y) =
      (case x*y of
         1 => "Ett"
       | 2 => "Två"
       | 3 => "Tre"
       | 4 => "Fyra"
       | x => Int.toString x) ^ "."
```

```
pratsamMultiplikation(2,3)
--> (case 2*3 of ... x => Int.toString x)^"."
--> (case 6 of ... x => Int.toString x)^"."
--> Int.toString 6 ^ "."
--> "6" ^ "."
--> "6."
```

Den nya bindningen av `x` *skuggar* den gamla.

Räckvidden av det *första* `x` omfattar *inte* `Int.toString x`.

Bindningsomgivningar

Ett annat sätt att förklara bindningar som bättre stämmer överens med vad som faktiskt händer i datorn är med *bindningsomgivningar*. Istället för att byta ut bundna variabler vid anrop (case-satser) håller man redan på bindningarna och byter när variabeln skall beräknas.

```
pratsamMultiplikation(2,3)
—>(case x*y of ...)^"."   [x —> 2, y —> 3]
—>(case 2*y of ...)^"."   [x —> 2, y —> 3]
—>(case 2*3 of ...)^"."   [x —> 2, y —> 3]
—>(case 6 of ...)^"."   [x —> 2, y —> 3]
—>(Int.toString x [x —> 6]) ^"." [x —> 2, y —> 3]
—>(Int.toString 6 [x —> 6]) ^"." [x —> 2, y —> 3]
—> "6"^"." [x —> 2, y —> 3]
—> "6."
```

Bindningsomgivningen `[x—> 2, y —> 3]` (inte en del av ML-syntaxen!) anger att `x` och `y` tillfälligt är bundna till heltalen `2` och `3`.

Bindningsordningen är viktig

Alla namn (inklusive funktioner) måste bindas *innan* de används.

Har man skrivit funktionen `add1` med definitionen

```
fun add1 x = x+1
```

och `add2` med definitionen

```
fun add2 x = add1(add1 x)
```

Så *måste* `add2` definieras *efter* `add1` – annars klagar ML på att `add1` är obunden.

```
- fun add2 x = add1(add1 x);
! Toplevel input:
! fun add2 x = add1(add1 x);
!               ^^^^
! Unbound value identifier: add1
```

Bindning är inte tilldelning

En ombindning av ett namn *skuggar* ev. tidigare bindingar av namnet, men *ändrar dem inte*.

```
- val x = 2;
> val x = 2 : int
- fun addx n = n+x
> val addx = fn : int -> int
- addx 0;
> val it = 2 : int
- val x = 4;
> val x = 4 : int
- addx 0;
> val it = 2 : int
```

`x` binds.

`x` är en *fri variabel* i `addx`.

`x` binds om...

men det påverkar inte `addx`!

`addx` använder den definition av `x` som gällde när `addx` själv definierades! Skall ombindningen av `x` märkas vid anrop av `addx` måste även `addx` definieras om.

Bindning är inte tilldelning (forts.)

Samma sak gäller ändring av funktionsdefinitioner.

```
- fun add1 x = x+2;
> val add1 = fn : int -> int
- fun add2 x = add1(add1 x);
> val add2 = fn : int -> int
- add2 3;
> val it = 7 : int
- fun add1 x = x+1;
> val add1 = fn : int -> int
- add1 3;
> val it = 4 : int
- add2 3;
> val it = 7 : int
```

...slant på tangentbordet...

...hoppsan...

Rättad definition.

Ok nu.

Fortfarande fel!

Även `add2` måste definieras om när `add1` ändras…

Lösning på detta problem: Ändra inte en funktion direkt i Moscow ML, utan lägg programmet i en fil och läs in hela filen efter ändring.

Funktionsvärda uttryck

Funktioner i ML är ”första klassens objekt” och kan *beräknas* med t.ex. if-then-else precis som andra värden. Alternativ `qcalc`:

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

```
qcalc((1,2),(1,3),#"+")
--> (if #"+"= #"+" then qadd else...)((1,2),(1,3))
--> (if true then qadd else ...) ((1,2),(1,3))
--> qadd((1,2),(1,3))
--> (5,6)
```

Anonyma funktioner

Man kan skriva funktionsuttryck *direkt* där de skall användas utan separat definition – en *anonym funktion*. Detta är ibland praktiskt.

```
(fn x => x*x) 4 ⟶ 4*4 ⟶ 16
```

Vi utvidgar `qcalc` med medelvärdesoperation (`a`), men utan att definiera någon separat funktion för beräkningen:

```
fun qcalc(x,y,opr) =
     (if opr = #"a" then
            fn (x,y) => qdiv(qadd(x,y),(2,1))
      else if opr = #"+" then
        qadd
      ……)  (x,y)
```

```
qcalc((1,2),(3,2),#"a") ⟶ (8,8)
```

(Argumentnamnen till den anonyma funktionen, `x` och `y`, är samma som i huvudfunktionen. Vilken är räckvidden? Problem? Lämpligt?)

Funktionstyper är inte likhetstyper

Man kan inte jämföra om två funktionsvärden med = eller <>!

Anledningen är att det kan vara mycket svårt – ja principiellt omöjligt – att avgöra om två funktioner är lika. Är t.ex.

```
(fn x => 2*x) = (fn x => if x<0 then x*2 else x+x)
```

…ja, det är det, men hur skall man kunna veta det?

(Att man i allmänhet inte kan avgöra ifall två funktioner är lika är ett fundamentalt resultat som Turing visade.)

Av liknande skäl kan man inte få ett funktionsvärde utskrivet – ML svarar bara `fn` – och med typen förstås.

```
- qcalc;
> val it = fn : (int*int)*(int*int)*char->int*int
```

Indentering

En god *indentering* är mycket viktigt för att få läsbara program!

```
fun namn mönster1 = uttryck1
  | namn mönster2 = uttryck2
      ...
  | namn mönsterN = uttryckN
```

```
if test1 then
  uttryck1
else if test2 then
  uttryck2
....
else
  uttryckN
```

```
case uttryck0 of
  mönster1 => uttryck1
| mönster2 => uttryck2
    ...
| mönsterN => uttryckN
```

Är uttrycken stora kan de skrivas indragna på nästa rad, t.ex.

```
fun namn mönster1 =
      uttryck1
```

```
case uttryck0 of
  mönster1 =>
    uttryck1
```

Kodningsstandard

I många sammanhang använder man en *kodningsstandard* som ger anvisningar för hur program skall skrivas.
I kursen har vi en kodningsstandard som består av två delar:

- Krav på att ha funktionsspecifikationer enligt tidigare beskrivning.
- Krav på att indentera program enligt föregående bild.

Syftet med kodningsstandarden är att

- Se till att grundläggande programdokumentation finns
- Underlätta för assistenterna när de skall rätta uppgifter
- Lära er arbeta med kodningsstandarder

Kodningsstandarden beskrivs på kurswebben
(välj ”Kodningsstandard” från menyn till vänster.)

Syntaktiskt socker

```
if B then E1 else E2
```

är *exakt samma sak* som

```
case B of true => E1 | false => E2
```

```
case E of P1=>E1 | P2=>E2 | ... | Pn=>En
```

är *exakt samma sak* som

```
(fn P1=>E1 | P2=>E2 | ... | Pn=>En) E
```

```
fn P1=>E1 | P2=>E2 | ... | Pn=>En
```

är *exakt samma sak* som

```
fn x => case x of P1=>E1 | P2=>E2 | ... | Pn=>En
```

Med syntaktiskt socker menas att man inför en form av uttryck som är ett enklare sätt att uttrycka något som redan finns i språket.

if-then-else är onödig och case och funktionsmatchning utbytbara!