UPPSALA
UNIVERSITET

Programmeringsmetodik DV1
Programkonstruktion 1

Moment 7
Om generella datastrukturer och träd

Typdeklarationer

Man kan ge namn åt typer, på ungefär samma sätt som man kan ge namn åt värden.

```
type rational = int*int;
type heltalslista = int list;
```

`type` är en *deklaration*, dvs den utför ingen beräkning utan talar om för ML hur program skall tolkas – i detta fall att `rational` och `heltalslista` är alternativa beteckningar för typerna `int*int` respektive `int list`.

Den huvudsakliga användningen av `type` är att ge mer läsbara program.

Vad används typerna till?

```
[((11,3),[(10,"Föreläsning PK1/PM1"),(15,"Avdelningsmöte"),
          (18,"Hämta Jonatan från ju-jutsun")]),
 ((11,4),[(10,"Föreläsning PK1/PM1"),(12,"Lunchseminarium"),
          (13,"Föreläsning PK1/PM1"),(19,"Möte Upsala Ö-gruppen")])
]
```

Typen är `((int*int)*(int*string) list) list`!?

Namn på de sammansatta typerna underlättar läsningen:

```
type month = int;
type day = int;
type date = month*day
type time = int;
type description = string;
type booking = time*description;
type daycalendar = date*booking list;
type calendar = daycalendar list;
```

Uttrycket ovan har nu typen `calendar`!

Namnabstraktion hjälper inte alltid

ML blir i praktiken otypat vad beträffar "likadana" namngivna typer.

```
type weekday = int;
val Mon = 1;
val Tue = 2;
……
val Sun = 7;
```

• Felaktiga mönster:

```
(* weekend d
   TYPE: weekday->bool
   PRE: d är en korrekt dag
   POST: true om d är en
         veckoslutsdag (lö/sö) *)
fun weekend Sat = true     ajaj.....
  | weekend Sun = true     ajaj.....
  | weekend _ = false;
```

• Felaktiga värden:

```
…weekend(8)…               ojoj.....
```

• Hopblandning av data:

```
(* overtimePay(d,x)
   TYPE: weekday*int->int
   PRE: d är en korrekt dag.
   POST: Timlönen en viss veckodag
         d med hänsyn till övertid
         om grundtimlönen är x. *)
fun overtimePay(d,x) =
  if weekend d then
    d * 2                  ujuj.....
  else
    x;
```

`Sat`, `Sun` är identifierare vilka som helst – de binds i mönster.
`weekday` är ett annat namn på `int` – inget typfel.

Uppräkningstyper

```
datatype weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
```

`weekday` blir en *helt ny* datatyp med de angivna värdena.

• Korrekt mönster:

```
(* weekend d
   TYPE: weekday->bool
   PRE: (inget)                    Obs!
   POST: true om d är en
         veckoslutsdag (lö/sö) *)
fun weekend Sat = true             Ok!
  | weekend Sun = true             Ok!
  | weekend _ = false;
```

• Felaktiga värden:

```
…weekend(8)…                       Typfel!
```

• Hopblandning av data:

```
(* overtimePay(d,x)
  TYPE: weekday*int->int
  PRE: (inget)                     Obs!
  POST: Timlönen en viss veckodag
        d med hänsyn till övertid
        om grundtimlönen är x. *)
fun overtimePay(d,x) =
  if weekend d then
    d * 2                          Typfel!
  else
    x;
```

`Sat`, `Sun` är *värden* som `true`, `false` – de binds *inte* i mönster.

`weekday` är *inte* samma typ som `int` –

sammanblandning ger typfel.

Konvertering av uppräkningstyper

```
datatype weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun;
```

ML definierar ingen speciell ordning av värdena i `weekday`.
Inte heller finns något inbyggt sätt att omvandla dem till/från heltal som man t.ex. kan med tecken (`ord`, `chr`). Sådant får man göra själv...

```
(* daynumber d
   TYPE: weekday->int
   PRE: (inget)
   POST: ordningstalet
         för veckodagen d *)
fun daynumber Mon = 1
  | daynumber Tue = 2
  | daynumber Wed = 3
  | daynumber Thu = 4
  | daynumber Fri = 5
  | daynumber Sat = 6
  | daynumber Sun = 7;
```

```
(* numberday n
   TYPE: int->weekday
   PRE: 1 <= n <= 7
   POST: veckodagen med
         ordningstal n *)
fun numberday 1 = Mon
  | numberday 2 = Tue
  | numberday 3 = Wed
  | numberday 4 = Thu
  | numberday 5 = Fri
  | numberday 6 = Sat
  | numberday 7 = Sun
  | numberday _ = raise Domain;
```

Vissa inbyggda typer kan definieras

```
datatype bool = true | false;
datatype unit = ();
```

Konstruerade (taggade) typer

```
type month = int;
type day = int;
type date = month*day;
type rational = int*int;
```
...ett rationellt tal.

Inget hindrar hopblandning av värden från `date` och `rational`.

Det är bättre att använda deklarationen `datatype`:

```
datatype date = Date of month*day;
datatype rational = Q of int*int;
```

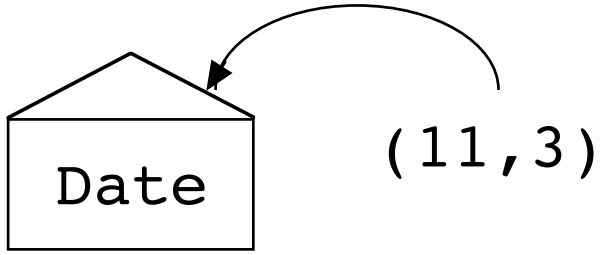
`date` och `rational` är nu *helt nya* datatyper som båda *innehåller* en `int*int`-tupel men som har en *etikett* (*tag*) som skiljer dem åt.

Värden av typen `date` skrivs föregångna av *konstruktorn* `Date`, t.ex. `Date(11,3)`. Samma med rationella tal, t.ex. `Q(11,3)`.

Jämför konstruktorn `::` som skapar en listcell.

Användning av konstruerade typer

Man kan betrakta en värde av en konstruerad datatyp som ett kuvert med konstruktorn påskriven, i vilken man har stoppat ned de data som ingår i värdet.



Försök att blanda ihop ett datum och ett rationellt tal ger typfel eftersom ML kan se på konstruktorerna att de hör till olika typer.

`Date` och `Q` har funktionstyper: `Date:int*int->date`
`Q:int*int->rational`

och ser ut och kan användas som funktioner men är inte funktioner i vanlig mening eftersom de inte utför någon beräkning. De skapar ett nytt värde på samma sätt som den infixa konstruktorn `::`.

En (delvis) taggad kalender

```
type month = int;
type day = int;
datatype date = Date of month*day
type time = int;
type description = string;
datatype booking = Booking of time*description;
datatype daycalendar =
                Daycalendar of date*booking list;
type calendar = daycalendar list;
```

```
[Daycalendar(Date(11,3),[Booking(10,"Föreläsning PK1/PM1"),
                         Booking(15,"Avdelningsmöte"),
                         Booking(18,"Hämta Jonatan från ju-
                                                    jutsun")]),
 Daycalendar(Date(11,4),[Booking(10,"Föreläsning PK1/PM1"),
                         Booking(12,"Lunchseminarium"),
                         Booking(13,"Föreläsning PK1/PM1"),
                         Booking(19,"Möte Upsala Ö-gruppen")])]
: calendar;
```

Konstruktorerna *får* vara samma som typnamnen – men normalt inte

Användning av konstruerade typer

För att komma åt delarna i ett värde av en konstruerad typ måste de tas ut ur kuvertet. För detta använder man matchning på liknande sätt som man matchar på listceller med `::`.

Nya värden av konstruerad typ kan skapas genom att ge uttryck som beräknas som argument till konstruktorn.

```
(* qadd(x,y)
   TYPE: rational*rational->rational
   PRE: x och y skall vara korrekta rationella tal
        (dvs nämnaren skall vara ≠ 0).
   POST: Summan av x och y
   EX: qadd(Q(1,2),Q(1,3)) = Q(5,6)
*)
fun qadd(Q(p1,q1),Q(p2,q2)) =
      Q(p1*q2+p2*q1,q1*q2);
```

Alternativa former för konstruerade typer

Formerna hos `datatype`-deklarationen för uppräkningstyper och konstruerade typen kan kombineras!

```
datatype number = Int of int | Real of real;
```

Ett värde i `number` är *antingen* ett heltal med etiketten `Int` eller ett flyttal med etiketten `Real`. Tack vare att etiketten skiljer kan man alltid veta om det handlar om ett heltal eller ett flyttal.

Användning:

- Om man kan ha värden med *olika form (utseende)*.
- Om man vill ha funktioner som kan ha *olika typer* som argument eller värde.

Typer med värden av olika form

Vi definierar en datatyp för att beskriva olika fordon. Vi anger för:

- bilar: reg.beteckning, färg och motorstyrka
- cyklar: färg och hjuldiameter
- skateboard: färg

```
datatype vehicle = Car of string*string*int
                 | Bicycle of string*int
                 | Skateboard of string;
```

Olika värden av typen `vehicle`:

```
Car("XYZ123","green",100)
Bicycle("red",26)
Skateboard "silver"
```

Exempel: räkna fordon av viss färg

```
(* vehicleColour vehicle
   TYPE: vehicle->string
   PRE: (ingen)
   POST: Färgen hos vehicle.
   EX: vehicleColour(Bicycle("green",24)) = "green" *)
fun vehicleColour (Car(_,colour,_)) = colour
  | vehicleColour (Bicycle(colour,_)) = colour
  | vehicleColour (Skateboard colour) = colour;

(* countColourVehicles(colour,vehicles)
   TYPE: string*vehicle list->int
   PRE: (ingen)
   POST: antal fordon i listan vehicles med
         färgen colour.
   EX: countColourVehicles("green", [Car("XYZ123","green",100),
                                     Skateboard "silver",
                                     Bicycle("green",24)]) = 2 *)

(* VARIANT: length vehicles *)

fun countColourVehicles(_,[]) = 0
  | countColourVehicles(colour,vehicle::rest) =
      (if vehicleColour vehicle = colour then 1 else 0) +
      countColourVehicles(colour,rest);
```

”Kombinerade” typer

Typen `number` kan innehålla ett heltal eller ett flyttal. Vid addition av `number` konverteras termerna automatiskt till flyttal om det behövs.

```
datatype number = Int of int | Real of real;

(* toReal n
   TYPE: number -> number
   PRE: (ingen)
   POST: Talet n konverterat till flyttal
   EX: toReal (Int 30) = Real 30.0 *)
fun toReal (Real r) = Real r
  | toReal (Int i) = Real (real i);

(* addNumbers(x,y)
   TYPE: number*number->number
   PRE: ingen
   POST: summman av x och y
   EX: addNumbers(Int 30, Real 2.0) = Real 32.0 *)
fun addNumbers(Int x, Int y) = Int (x+y)
  | addNumbers(x,y) = let val Real xr = toReal x;
                          val Real yr = toReal y
                      in  Real (xr+yr)
                      end;
```

Polymorfa datatyper

En eller flera komponentdatatyper i en konstruerad datatyp kan vara en typvariabel:

```
datatype 'a option = NONE | SOME of 'a
```

`option` blir här en *typ*konstruktor precis som `list`.

Instansen `int option` är en typ med två slags värden:

- Konstanten `NONE`.
- Ett heltal taggat med konstruktorn `SOME`.

Exempel på värden av typen `int option`:

```
NONE, SOME 4, SOME ~2, SOME 0
```

Exempel på värden av typen `string option`:

```
NONE, SOME "Hej", SOME "", SOME "ABC"
```

Användning av `option`

`option` är en inbyggd typkonstruktor i ML. Den används t.ex. när man vill ha ett bättre sätt att hantera felsituationer i programmet än att avbryta körningen eller göra något odefinierat.

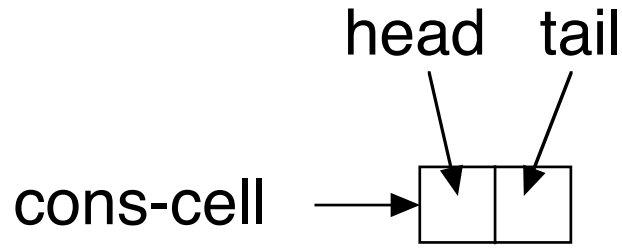
```
(* sumUpTo n
   Type: int->int option
   Pre: (ingen)
   Post: SOME s, där s är summan av talen 0...n om n>=0.
         NONE annars.
   Ex: sumUpTo 4 = SOME 10
       sumUpTo ~2 = NONE *)
(* Variant: n *)
fun sumUpTo n = if n < 0 then
                  NONE
                else if n = 0 then
                  SOME 0
                else
                  SOME(n+valOf(sumUpTo(n-1)));
```

Förvillkor saknas – jämför med `sumUpTo` från kursmoment 4!

(`valOf` är en inbyggd funktion i ML: `fun valOf(SOME x) = x`)

Rekursiva typer

Komponenterna i en konstuerad datatyps värde får höra till datatypen själv! (En *rekursiv* typ.) En listcell



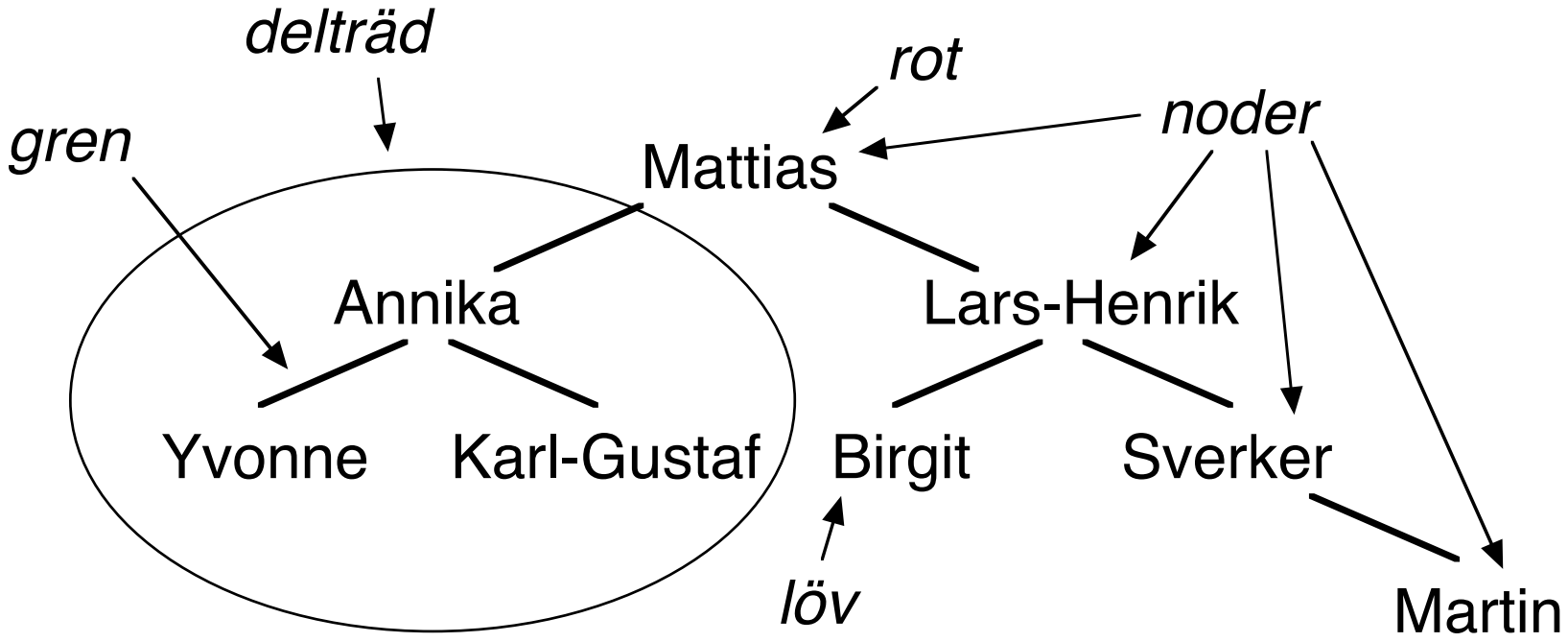
kan definieras som en konstruerad datatyp som innehåller en tupel med två komponenter – för head resp. tail:

```
datatype 'a list = [] | :: of 'a*'a list;
```

Listor i ML fungerar exakt som om de var definierade med denna deklaration.
(Man kan inte göra om deklarationen för att prova – ML tillåter inte att man gör en ”ny” deklaration av listor.)

Träd

Listor kan representera data med linjär struktur – t.ex. förteckningar och sekvenser. Ofta har dock data *trädstruktur*, t.ex. i ett släktträd:



Detta är ett *binärt träd* – varje nod har *maximalt* två *grenar*.
Löv är noder som saknar grenar. *Roten* är trädets början.
Djupet hos en nod är avståndet till roten.

Hur representera ett träd

Varje nod läggs i en lista tillsammans med delträdens rotnoder:

```
type familyTree = (string*string*string) list;

[("Mattias","Annika","Lars-Henrik"),
 ("Annika", "Yvonne", "Karl-Gustaf"),
 ("Yvonne", "", ""),
 ("Karl-Gustaf", "", ""),
 ("Lars-Henrik", "Birgit", "Sverker"),
 ("Sverker", "", "Martin"),
 ("Birgit", "", ""),
 ("Martin", "", "")] : familyTree
```

+ Enkel representation
– ”Navigering” i trädet kräver *sökning* i listan. Hur hitta farfar?
– Vilken nod är rotnoden?
– Uppdatering av trädet kräver också sökning.

Krånglig navigering

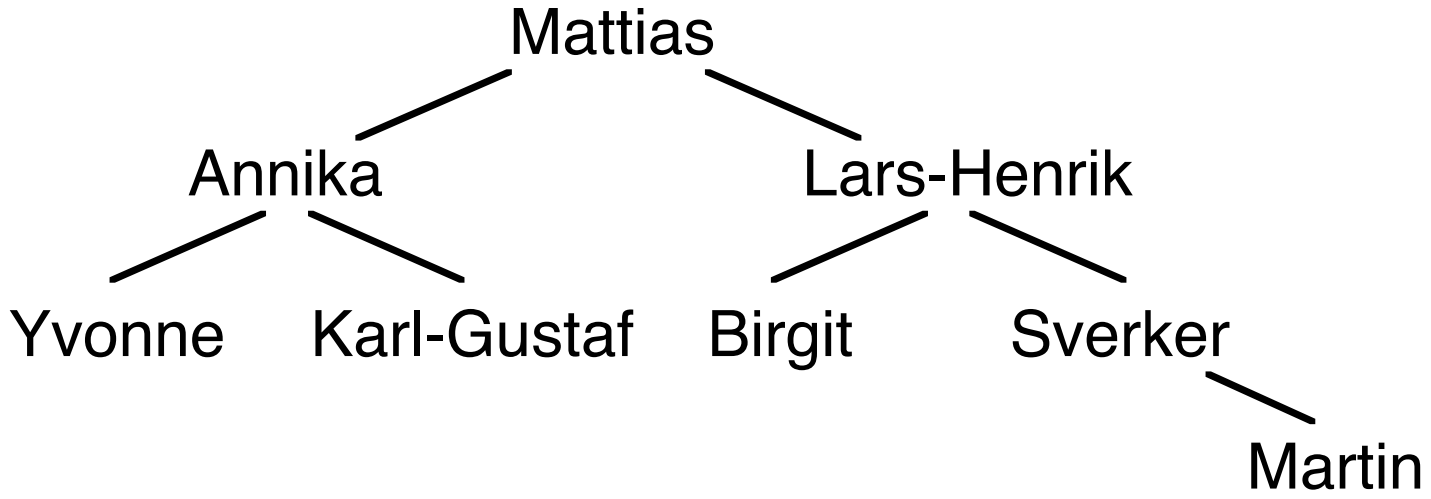
```
(* searchTree(person,tree)
   TYPE: string*familyTree->string*string*string
   PRE: Trädet tree innehåller en nod med personen person.
   POST: Nodinformationen för person.
   EX: searchTree(["Lars-Henrik",
                   [...,("Lars-Henrik", "Birgit", "Sverker"),...]
        = ("Lars-Henrik", "Birgit", "Sverker") *)
(* VARIANT: length tree *)
fun searchTree(key,nod::rest) = if #1 nod = key then
                                  nod
                                else
                                  searchTree(key,rest);

(* farfarsnamn t
   TYPE: familyTree -> familyTree
   PRE: Trädet t innehåller farfadern till rotnoden.
   POST: Namnet på rotnodens farfar.
   EX: farfarsnamn([("Mattias","Annika","Lars-Henrik"),....
                    ("Lars-Henrik", "Birgit", "Sverker"),...]
        = "Sverker" *)
fun farfarsnamn((_,_,far)::rest) = #3 (searchTree(far, rest));
```

Träddatatyper

Binära träd kan representeras *direkt* i ML som en rekursiv datatyp!

```
datatype familyTree = Empty
            | Person of string*familyTree*familyTree;
```




```
Person("Mattias",
       Person("Annika",Person("Yvonne",Empty,Empty),
                       Person("Karl-Gustaf",Empty,Empty)),
       Person("Lars-Henrik",Person("Birgit",Empty,Empty),
                            Person("Sverker",Empty,
                                             Person("Martin",
                                                    Empty,Empty))))
```

Enkel navigering

```
(* farfar t
   TYPE: familyTree -> familyTree
   PRE: Det finns en farfader till
        rotnoden i t.
   POST: Delträdet till t där farfadern är rotnod.
   EX: farfar
         (Person("Mattias",…,
                 Person("Lars-Henrik",…,
                        Person("Sverker",…,…))))=
       Person("Sverker",…,…)
*)
fun farfar(Person(_,_,Person(_,_,ff))) = ff;
```

Rekursion över träd

Rekursion över träd är normalt *dubbel* rekursion – man måste göra rekursion över *båda grenarna* från en given nod.

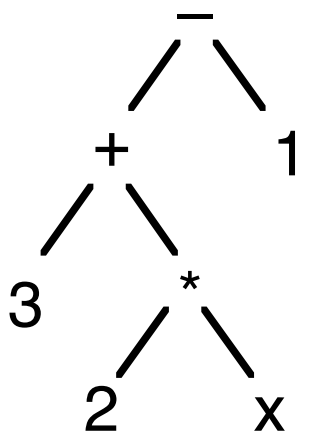
```
(* countPersons t
   TYPE: familyTree -> int
   POST: Antal personer i trädet t *)
(* VARIANT: Antal noder i trädet t *)
fun countPersons Empty = 0
  | countPersons(Person(_,Left,Right)) =
       1+countPersons Left+countPersons Right;
```

```
countPersons(Person("Ronja",Person("Lovis",Empty,Empty),
                            Person("Mattis",Empty,Empty)))
--> 1+countPersons(Person("Lovis",Empty,Empty)+
      countPersons(Person("Mattis",Empty,Empty))
--> 1+(1+countPersons Empty+countPersons Empty)+
      countPersons(Person("Mattis",Empty,Empty))
--> 1+(1+0+countPersons Empty)+
      countPersons(Person("Mattis",Empty,Empty))
--> 1+(1+countPersons Empty)+
      countPersons(Person("Mattis",Empty,Empty))
--> 1+(1+0)+countPersons(Person("Mattis",Empty,Empty))
--> 2+countPersons(Person("Mattis",Empty,Empty))
--> 2+(1+countPersons Empty+countPersons Empty)     ..... --> 3
```

Syntaxträd

Strukturen hos en formel kan beskrivas med ett *syntaxträd*.

3+2*x-1 :




```
datatype expr = Const of int
              | Var of string
              | Plus of expr*expr
              | Minus of expr*expr
              | Times of expr*expr
              | Div of expr*expr;
```

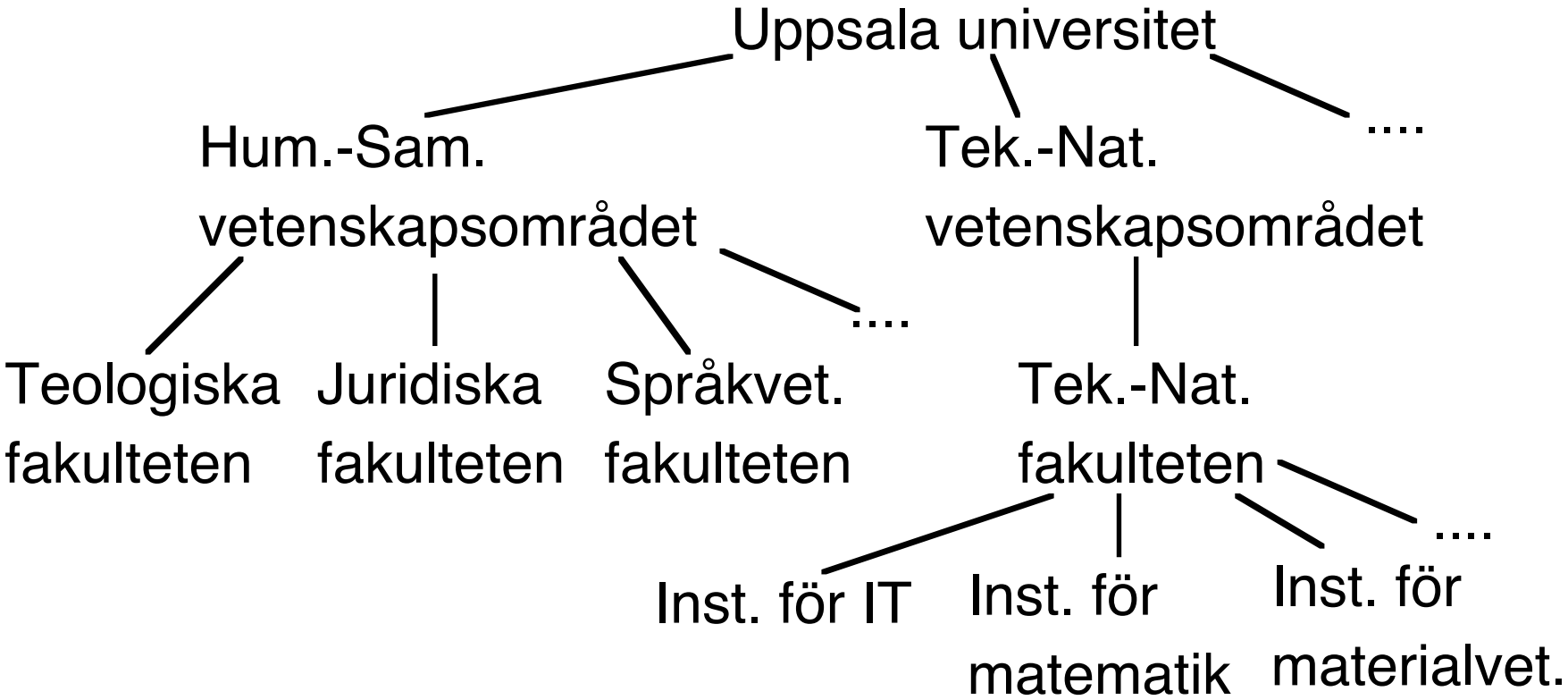
```
Minus(Plus(Const 3,Times(Const 2,Var "x")),
      Const 1)
```

Beräkning av uttryck

```
exception UnDefVar of string;

(* eval(expr,var,value)
   TYPE: expr*string*int->int
   PRE: expr innehåller inga variabler utom den med namn var.
        Ingen division med 0 förekommer i expr.
   POST: Värdet av expr uträknat om
         variabeln med namn var har värdet value.
   Ex: eval(Minus(Plus(Const 3,Times(Const 2,Var "x")),Const 1),
            "x",4) = 10 *)
(* VARIANT: Storleken hos expr. *)
fun eval(Const c,_,_) = c
  | eval(Var v,var,value) = if v = var then
                              value
                            else
                              raise UnDefVar v
  | eval(Plus(e1,e2),var,value) = eval(e1,var,value) +
                                  eval(e2,var,value)
  | eval(Minus(e1,e2),var,value) = eval(e1,var,value) -
                                   eval(e2,var,value)
  | eval(Times(e1,e2),var,value) = eval(e1,var,value) *
                                   eval(e2,var,value)
  | eval(Div(e1,e2),var,value) = eval(e1,var,value) div
                                 eval(e2,var,value);
```

Allmänna träd




```
datatype 'a tree = Tree of 'a*('a tree) list;
```

'a är typen för informationen i varje nod.

Exempel på träd

```
datatype 'a tree = Tree of 'a*('a tree) list;

Tree("Uppsala universitet",
     [Tree("Hum.-Sam. vetenskapsområdet",
           [Tree("Teologiska fakulteten",[]),
            Tree("Juridiska fakulteten",[]),
            Tree("Språkvet. fakulteten",[]),
            ...]),
      Tree("Tek.-Nat. vetenskapsområdet",
           [Tree("Tek.-Nat. fakulteten"),
                 [Tree("Inst. för IT",[]),
                  Tree("Inst. för matematik",[]),
                  Tree("Inst. f. materialvet",[]),
                  ...])]),
      ...]) : string tree
```

Felkoder (exceptions)

Felkoder är egentligen konstruktorer för datatypen `exn`.

Deklareras med deklarationen `exception`.

```
exception NegArg;
fun sumUpTo 0 = 0
  | sumUpTo n = if n < 0 then
                  raise NegArg
                else
                  n+sumUpTo(n-1);
```

Anropet `sumUpTo ~2;`
Ger:
```
! Uncaught exception:
! NegArg
```

```
exception Error of string;
fun sumUpTo' 0 = 0
  | sumUpTo' n = if n < 0 then
                   raise Error("sumUpTo' "^Int.toString(n))
                 else
                    n+sumUpTo'(n-1);
```

Anropet `sumUpTo ~2;`
Ger:
```
! Uncaught exception:
! Error  "sumUpTo' ~2"
```