



UPPSALA  
UNIVERSITET

# Programkonstruktion och datastrukturer


## Moment 9

### Abstrakta datatyper

# Tabeller igen

Vi har sett olika exempel på hur man kan implementera tabeller – att ifrån en ”nyckel” kunna ta fram ett ”tabellvärde”.

<i>Bol, Roland</i>	7606
<i>Eriksson, Lars-Henrik</i>	1057
<i>Flener, Pierre</i>	1028



nyckel                      värde

I de fortsatta exemplen kommer jag att förutsätta att nycklarna är strängar, men inte förutsätta något om tabellvärdenas typer.

Antag att tabellernas datatyp har namnet `'a table` där `'a` är typen för tabellvärdena, t.ex. `int table` i ovanstående exempel.

# Operationer på tabellen

Tänk på tabellen ur ett *användarperspektiv*.

Vad vill man kunna göra för beräkningar med tabeller?

- Lägg in en uppgift i en tabell (`insert`)
- Kontrollera om en nyckel finns i en tabell (`exists`)
- Hämta ett värde från en tabell (`search`)
- Tag bort en uppgift från en tabell (`delete`)

(Fler operationer är tänkbara, vi kommer att diskutera några till, men nöjer oss med dessa just nu.)

Not: Man skriver lite slarvigt att man "lägger in" eller "tar bort" uppgifter, men tabeller representeras förstås av datastrukturer som all annan information. `insert` och `delete` beräknar alltså nya tabeller med förändrat innehåll, snarare än ändrar den gamla.

# Specifikationer för tabellfunktionerna

(\* insert(T,k,v)

TYPE: 'a table\*string\*'a -> 'a table

PRE: T är en korrekt tabell.

POST: Tabellen T uppdaterad med värdet  
v för nyckeln k \*)

(\* exists(T,k)

TYPE: 'a table\*string -> bool

PRE: T är en korrekt tabell.

POST: true om nyckeln k finns i tabellen T,  
false annars. \*)

(\* search(T,k)

TYPE: 'a table\*string -> 'a

PRE: T är en korrekt tabell.

Nyckeln k finns i tabellen T.

POST: Värdet som hör ihop med nyckeln k  
i tabellen T. \*)

# Specifikationer för tabellfunktionerna (forts.)

(\* delete(T,k)

TYPE: 'a table\*string -> 'a table

PRE: T är en korrekt tabell.

POST: Tabellen T med uppgiften om nyckeln  
k borttagen \*)

Innan man kan börja arbeta med tabeller så måste man ha en första tabell att utgå ifrån.

Låt oss alltså anta att det finns ett *namngivet värde* `empty` som är en tom tabell.

# Användning av tabellen

Vi låter någon programmera tabellfunktionerna åt oss och sedan kan vi skriva program som använder tabeller!

```
val datalogi =  
  insert(insert(insert(empty,  
                      "Eriksson, Lars-Henrik",1057),  
          "Flener, Pierre",1028),  
        "Bol, Roland",7606);
```

```
search(datalogi,"Eriksson, Lars-Henrik") —> 1057  
exists(datalogi,"Eriksson, Lars-Henrik") —> true  
search(insert(datalogi,"Bol, Roland",1234),  
        "Bol, Roland") —> 1234  
exists(delete(datalogi,"Flener, Pierre"),  
        "Flener, Pierre") —> false
```

Det verkar fungera bra, men fattas inte något...?

# Abstrakta datatyper

Vi har inte sagt någonting om *hur* tabellerna skall *representeras*.

Kunskapen om hur tabeller ser ut finns *i koden* för funktionerna `exists`, `search`, `insert` och `delete` samt i värdet `empty`.

Den som skriver funktionerna måste veta hur tabeller representeras, men *inte* den som *använder* funktionerna. Den del av programmet som använder tabeller behöver inte "titta inuti" dem.

Tabellen kan representeras som lista, ett sökträd eller något annat.

*Användningen* av tabeller är alltså *oberoende* av representationen. Man säger att representationen av tabellen har *abstraherats bort* genom *dataabstraktion* – tabellen är en *abstrakt datatyp* (ADT).

# Gränsyta

Namn och typ för funktionerna `exists`, `search`, `insert` och `delete` samt värdet `empty` är den abstrakta datatypens *gränsyta* (eng. *interface*) mot kod som *använder* tabeller.

En ADT kan ses som en "svart låda" där bara gränsytan är synlig utåt. Datastrukturen, ev. hjälpfunktioner m.m. är dolda.

```
empty: 'a table
insert: 'a table*string*'a -> 'a table
exists: 'a table*string -> bool
search: 'a table*string -> 'a
delete: 'a table*string -> 'a table
```



# En implementering av tabellen

Ett sätt att implementera tabellen är som en lista.

```
datatype 'a table = Table of (string*'a) list;
```

```
val empty = Table [];
```

```
fun search(Table ((k',v)::l),k) =  
    if k'=k then  
        v  
    else  
        search(Table l,k);
```

...etc...

datalogi —>

```
Table [("Bol, Roland",7606),  
      ("Flener, Pierre",1028),  
      ("Eriksson, Lars-Henrik",1057)] : int table;
```

# En annan implementering av tabellen

Ett annat sätt att implementera tabellen är som ett (obalanserat) binärt sökträd.

```
datatype 'a table = Void
                | Bst of (string*'a)*'a table*'a table;
val empty = Void;
```

```
fun search(Bst((k',v),L,R), k) =
    if k = k' then v
    else if k < k' then search(L, k)
    else search(R, k)
```

...etc...

datalogi —>

```
Bst ("Eriksson, Lars-Henrik",1057),
    Bst ("Bol, Roland",7606),Void,Void),
    Bst ("Flener, Pierre",1028),Void,Void)): int table;
```

# Byte av representation

En implementering av en ADT kan utan vidare bytas mot annan med andra algoritmer och/eller datastrukturer. Är gränsytan densamma så påverkas inte resten av programmet.

```
search(insert(datalogi, "Bol, Roland", 1234),  
        "Bol, Roland")
```

använder



```
empty: 'a table  
insert: 'a table*string*'a -> 'a table  
exists: 'a table*string -> bool  
search: 'a table*string -> 'a  
delete: 'a table*string -> 'a table
```

Tabell-ADT med listor  
Table syns inte

fungerar



```
empty: 'a table  
insert: 'a table*string*'a -> 'a table  
exists: 'a table*string -> bool  
search: 'a table*string -> 'a  
delete: 'a table*string -> 'a table
```

Tabell-ADT med sökträd  
Void och Bst syns inte



Byts mot

# Fulkodning

Principiellt är ADTer svarta lådor, men det går att "öppna locket".

Den som vet att tabellen är implementerad som en lista kan frestas skriva en egen funktion – utanför ADTn – för att lägga till *nya* nycklar i listan på detta sätt:

```
fun add(Table table,k,v) = Table ((k,v)::table);
```

Detta

- bryter abstraktionen
- gör att man riskerar programfel vid byte av implementering av ADTn
- kan också göras av misstag

# Stöd för dataabstraktion i ML

För att förhindra att användaren av en abstrakt datatyp av misstag (eller avsiktligt...) gör sig beroende av representationen så har ML stöd för att tvinga användningen att bli representationsoberoende. `abstype`-deklarationen är en variant av `datatype`-deklarationen.

```
abstype datatype-deklaration
  with
    deklarerationer av gränsytan
end;
```

Operationer i den abstrakta datatypens gränsyta definieras mellan `with` och `end`. Konstruktorerna i datatypsdeklarationen blir *osynliga* och *oåtkomliga* utom just mellan `with` och `end`. Detta tvingar datatypen att bli abstrakt.

Av samma skäl skrivs inte `abstype`-värden ut av ML-systemet.

# Tabellen som abstype

```
abstype 'a table = ...  
with  
  val empty = ...;  
  fun insert(...) = ...;  
  fun exists(...) = ...;  
  fun search(...) = ...;  
  fun delete(...) = ...;  
end;
```

# Olika sorters operationer i gränsytan

Man brukar skilja på

- Operationer som konstruerar nya värden av den abstrakta datatypen (t.ex. `insert`). De kallas ofta *konstruktorer*. (Men blanda inte ihop dem med konstruktorer som används för att tagga data – t.ex. `Table`).
- Operationer som hämtar delar av värden (t.ex. `search`). De kallas ofta *selektorer*.
- Operationer som gör tester eller jämförelser på värden av den abstrakta datatypen (t.ex. `exists`). De kallas ofta *predikat*.

Gränserna mellan dessa är dock inte skarp.

# Intern dokumentation av abstrakta datatyper

Varje `abstype` skall ha intern dokumentation på samma sätt som en `datatype`.

Dessutom skall alla definitioner av värden (`val`) som ingår i gränsytan också dokumenteras med

- Namn på värdet
- Värdets typ
- Beskrivning av vad värdet representerar

T.ex.:

```
(* empty
  TYPE: 'a table
  VALUE: En tom tabell *)
val empty = Table [];
```



# Fördelar med dataabstraktion

- Operationerna kan implementeras oberoende av användningen av den abstrakta datatypen. Den som skall använda den abstrakta datatypen *behöver* inte veta hur datatypen är implementerad.
- Eftersom den som använder den abstrakta datatypen inte heller *får utnyttja* eventuell kunskap om hur datatypen är implementerad så kan man ändra implementeringen utan att behöva ändra någonting i övriga delar av programmet. (Så länge som gränsytans funktionsspecifikationer följs!)
- Av samma skäl så behöver den som skall använda datatypen inte bry sig om att upprätthålla eller kontrollera (t.ex. i förvillkor) invarianter – det vilar helt på ADT-koden.
- Den tydliga gränsen mellan implementering och användning gör det lättare att *återanvända* ADTn i andra situationer.

# Nackdelar med dataabstraktion

- Man kan bara göra sådana operationer som är förutsedda.  
Exempel: Om man vill veta om tabellen är *tom*, så går inte det!
- Man kan förlora prestanda i vissa fall.  
Exempel: Att lägga till en *ny* nyckel (som inte fanns förut) i en tabell som representeras som en lista *går* att göra i konstant tid. (Titta på "fulkoden" på tidigare sida.) Exempel-ADTn erbjuder bara funktionen `insert` som tar tid proportionell mot antal nycklar i tabellen.
- Speciellt för ML: Eftersom konstruktorerna inte går att använda utanför `abstype`-deklarationen så kan de inte används i matchning utanför. Det *kan* ge mer svårlästa program.

# Tänk igenom gränsytan

För att undvika problemen med att

- man kan bara göra sådana operationer som är förutsedda.
- man kan förlora prestanda i vissa fall.

krävs att man tänker igenom noga hur den abstrakta datatypen skall användas så att man har lämpliga operationer i gränsytan.

Försök att ha generella operationer som kan användas till mycket.

# Nödutgång

För att möjliggöra saker som man inte tänkt på i förväg kan gränsytan innehålla en "nödutgång" i form av en operation som "exporterar" innehållet i en ADT i ett bestämt format – t.ex. som en lista.

Exempel:

```
(* toList T
   TYPE: 'a table -> (string*'a) list
   PRE:  T är en korrekt tabell.
   POST: Tabellen som en lista av
         nyckel-värde-par. *)
```

Denna funktion är trivial att skriva i listimplementeringen eftersom tabellen redan ser ut så, men inte svår i sökträdsimplementeringen heller. (Använd en av traverseringsalgoritmerna.)

# Iteratorer

I gränsytan kan vara bra att ha en operation på en tabell som tillåter att man går igenom varje rad i tabellen och gör en beräkning.

Exempel:

```
(* iterate f e T
  TYPE: (string*'a*'b->'b)->'b->'a table -> 'b
  PRE:  T är en korrekt tabell.
  POST: f(k1,v1,f(k2,v2,...f(kn,vn,e)...)), där
        (k1,v1),(k2,v2),...,(kn,vn) är alla par av
        nycklar och värden i T i någon ordning. *)
```

I listimplementeringen av tabeller kan man definiera `iterate` som

```
fun iterate f e (Table []) = e
  | iterate f e (Table ((k,v)::l)) =
    f(k,v,iterate f e (Table l));
```

Observera likheten med `foldr`!

# Användning av iteratorer

Med iteratorer kan man enkelt göra många beräkningar med tabeller utan att behöva ha speciella funktioner för det i gränsytan.

Exempel: Räkna antalet element i en tabell:

```
(* countLines T
   TYPE: 'a table -> int
   POST: Antal rader i tabellen T *)
fun countLines T =
  iterate (fn (_,_,n) => n+1) 0 T;
```

Exempel: Summera alla tabellvärden i en int table

```
(* sumValues T
   TYPE: int table -> int
   POST: Summan av tabellvärdena i tabellen T *)
fun sumValues T =
  iterate (fn (_,x,y) => x+y) 0 T;
```

# Likhetstyper

Abstrakta datatyper i ML är *inte likhetstyper*. Värden med samma "beteende" kan ha olika representation!

Exempel:

```
val tab1 = insert(insert(empty, "B", 2), "A", 1);  
val tab2 = insert(insert(empty, "A", 1), "B", 2);
```

tab1 och tab2 är nu bundna till tabeller med *samma information*.

Men  $tab1 \neq tab2$ , därför att listorna har elementen i *olika ordning*.

För att förhindra misstag och för att inte fakta om datarepresentationen skall avslöjas *tillåter inte* ML likhetsjämförelser mellan värden av abstrakta datatyper utanför ADT-deklarationen.

För att kunna jämföra måste gränsytan ha en operation för jämförelsen som då kan ta hänsyn till representationen.

# Rationella tal som abstrakta datatyper

```
abstype rational = Q of int*int  
with
```

```
  fun qadd(Q(p1,q1),Q(p2,q2)) =  
      Q(p1*q2+p2*q1,q1*q2);
```

```
  fun qsub(Q(p1,q1),Q(p2,q2)) =  
      Q(p1*q2-p2*q1,q1*q2);
```

```
  fun qmul(Q(p1,q1),Q(p2,q2)) =  
      Q(p1*p2,q1*q2);
```

```
  fun qdiv(Q(p1,q1),Q(p2,q2)) =  
      Q(p1*q2,q1*p2);
```

...

Räcker detta?



# Typkonverteringar

Vi måste ha ett sätt att konvertera data till och från rationella tal! Det går inte längre att skapa ett rationellt tal genom att bara skriva t.ex.  $Q(5, 1)$ !

```
(* toRational n
   TYPE: int -> rational
   POST: n som ett rationellt tal *)
fun toRational n = Q(n,1);

(* fromRational x
   TYPE: rational -> real
   POST: Det rationella talet x som ett flyttal *)
fun fromRational(Q(p,q)) = real p / real q;
```

Räcker det nu då?

# Jämförelser

Vi måste ha ett enkelt sätt att jämföra rationella tal:

```
(* qequal(x,y)
   TYPE: rational*rational -> bool
   POST: true om x och y är samma rationella tal,
         false annars. *)
fun qequal(Q(p1,q1),Q(p2,q2)) = p1*q2 = p2*q1;

(* qless(x,y)
   TYPE: rational*rational -> bool
   POST: true om x är ett mindre rationellt tal
         än y, false annars. *)
fun qless(Q(p1,q1),Q(p2,q2)) = p1*q2 < p2*q1;

Man kan tänka sig fler operationer, men vi kan vara rätt nöjda nu.

end;
```

# Datatypen order

Ibland behöver man kunna jämföra två objekt både för storlek och likhet. Det finns en fördefinierad datatyp `order` för att beskriva resultatet av sådana jämförelser:

```
datatype order = LESS | GREATER | EQUAL;
```

Flera biblioteksfunktioner har värde av typ `order`:

```
Int.compare, String.compare, Real.compare, ...
```

```
Int.compare(3,3) = EQUAL
```

```
String.compare("Aha", "Ahaaaa!") = LESS
```

```
Real.compare(46.4, 32.0) = GREATER
```

Speciellt för abstrakta datatyper kan det vara praktiskt att ha en `order`-värd jämförelsefunktion i gränsytan.

# Exempel på order-värd jämförelse

```
(* qcompare(x,y)
   TYPE: rational*rational -> order
   POST: EQUAL om x och y är tal, LESS om x är
         mindre än y, GREATER annars. *)
fun qcompare(Q(p1,q1),Q(p2,q2)) =
  case p1*q2-p2*q1 of
    0 => EQUAL
  | d => if d < 0 then LESS
        else GREATER;
```

```
qcompare(Q(7,14),Q(6,9))
—> case 7*9-14*6 of 0 => EQUAL | d => if ...
—> case ~21 of 0 => EQUAL | d => if ...
—> if ~21 < 0 then LESS else GREATER
—> if true then LESS else GREATER
—> LESS
```

(Detta visar hur beräkningen sker. Eftersom `rational` är abstrakt kan man inte faktiskt skriva `qcompare(Q(7,14),Q(6,9))` till ML.)

# En programkonstruktionsdetalj

Man kan inte utan vidare göra ändra en funktion till att bli operation i en abstrakt datatyp eftersom datatypen har en "etikett" ("tag"/konstruktor), som kan vara ivägen.

Tag som exempel `insert` för tabeller representeras som listor:

```
fun insert([],k,v) = [(k,v)]  
  | insert((k',v')::T,k,v) =  
    if k'=k then  
      (k,v)::T  
    else  
      (k',v')::insert(T,k,v);
```

# En programkonstruktionsdetalj (forts.)

Nu vill vi göra tabellen till en abstrakt datatyp med deklarationen

```
datatype 'a table = Table of string * 'a =  
with
```

För att göra insert till en operation på table så måste listan  
”taggas” överallt.

```
fun insert(Table [],k,v) = Table [(k,v)]  
  | insert(Table ((k',v')::T),k,v) =  
    if k'=k then  
      Table((k,v):: T)  
    else  
      Table((k',v')::insert(Table T,k,v));
```

typfel

Vad är problemet?

# En programkonstruktionsdetalj (forts.)

En lösning:

```
fun insert(Table [],k,v) = Table [(k,v)]  
  | insert(Table ((k',v')::T),k,v) =  
    if k'=k then  
      Table((k,v):: T)  
    else  
      Table((k',v')::  
        let  
          val Table t = insert(Table T,k,v)  
        in  
          t  
        end);
```

# En programkonstruktionsdetalj (forts.)

En alternativ lösning med ursprungliga `insert` som *hjälpfunktion*:

```
fun insert(Table T,k,v) =  
  let  
    fun insert'([ ],k,v) = [(k,v)]  
      | insert'((k',v')::T,k,v) =  
        if k'=k then  
          (k,v)::T  
        else  
          (k',v')::insert'(T,k,v);  
  in  
    Table(insert'(T,k,v))  
  end;
```

Eftersom räckvidden för definitionen av `insert'` inte sträcker sig utanför definitionen av `insert`, så kommer `insert'` inte att ingå i den abstrakta datatypens gränsyta.



# Felsökning av abstrakta datatyper

I flera implementeringar av ML (dock *inte* Poly/ML) så döljer ML-systemet värden av sådana typer även för programmeraren:

```
- val x = toRational 3;
```

```
val x = - : rational
```

När man testar och felsöker funktionerna i en ADT kan det vara bra att *tillfälligt* göra datatypen oabstrakt genom att byta `abstype` mot `datatype` och ta bort `with` samt `end`.

```
datatype rational = Q of int*int
```

```
fun qadd.....
```

```
- val x = toRational 3;
```

```
val x = Q (3,1) : rational
```

Glöm bara inte att ändra tillbaka till `abstype` efteråt och kontrollera att programmet fortfarande fungerar!!

# Sammanfattning av abstraktion

*Definitionsabstraktion* – ett värde ersätts av ett symboliskt namn

Ex.: `x < maximum` i stället för `x < 100`

(vitsen är bättre läsbarhet och att behöver man ändra `maximum` räcker det med att ändra på ett ställe – där `maximum` definieras.)

*Funktionsabstraktion* – ett uttryck görs oberoende av specifika data

Ex.: `(fn x => x+1) y` i stället för `y+1`

(vitsen är att `(fn x => x+1)` kan namnges och användas i många olika sammanhang – ofta även bättre läsbarhet.)

*Dataabstraktion* – ett program görs oberoende av specifik datarepresentation

Ex.: `insert(table, x, y)` i stället för `(x, y)::table`

(vitsen är att man kan förändra representationen utan att ändra programmet, lättare att uppfylla invarianter, ofta bättre läsbarhet.)