UPPSALA UNIVERSITET

Programmeringsmetodik DV1
Programkonstruktion 1

Moment 6
Om listor och polymorfa typer

Program- och datastrukturer

	Programstruktur	Datastruktur
Sammansättning:	uttryck som argument	tupler
Alternativ:	if/case/matchning	?? (inte ännu)
Upprepning:	rekursion	*listor* (strängar)

dagens ämne

Listor

En *lista* är en *ordnad* uppsättning (*sekvens*, *följd*) av *element*, som kan ha godtycklig typ.

element

```
· [18, 12, ~5, 12, 10] : int list
· [2.0, 5.3, 3.7, ~1E5] : real list
· ["Lars", "Henrik", "PM1"] : string list
· [(1,"A"), (2,"B")] : (int*string) list
· [Math.sin, fn x=>x+1.0] : (real->real) list
· [[1], [2, 3]] : int list list
```

Karakteristiskt för listor:

- Alla element i en viss lista måste ha *samma typ*.
- En lista kan innehålla *godtyckligt många* element (eller *inga alls*).
- Samma element kan finnas *flera gånger* i samma lista.

Praktisk användning av listor

En förteckning (böcker i ett bibliotek):

```
["Introduction to Programming Using Standard ML",
 "Mathematical Logic", "Software Engineering",
 "Applications of Formal Methods"] : string list
```

En tabell (telefonanknytningar på inst. för IT):

```
[("Lars-Henrik Eriksson", 1057), ("Jesper Bengtsson", 3156),
 ("Mattias Wiggberg", 3176), ("Anna Sandström", 7595)]
 : (string*int) list
```

En kalender:

```
[((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")])
] : ((int*int)*(int*string) list) list
```

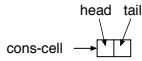

Listuttryck

Precis som tupler kan listor konstrueras av uttryck som beräknar ett visst värde.

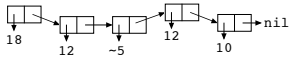
```
· [18, 3+9, 5-7, size("abc"), 10]
  ==> [18, 12, ~2, 3, 10]
· ["A"^"B", Int.toString(5)]
  ==> ["AB", "5"]
· [(1+1,2), qplus((1,2),(1,3))]
  ==> [(2,2), (5,6)]
```

En *tom lista*: `[]`

Struktur hos listor



Listor byggs upp som kedjor. Ex. `[18, 12, ~5, 12, 10]`



Man kommer bara åt element i listan genom att gå igenom länkarna.

Att komma åt ett element kräver alltså en tidsåtgång (antal steg) som är proportionell mot elementets plats i listan.

Pilarna (*pekarna*) kan inte ändras. Listor måste byggas *bakifrån*!
I ML är pekarna *implicita*, i en del andra språk (C) är de *explicita*.

Grundläggande operationer på listor.

`hd` (head) ger *första elementet* i en lista..

```
hd [1,2,3] ==> 1          (Resultatet blir inte [1])
hd [] ger exekveringsfel (Empty).
```

`tl` (tail) ger en lista *utom* första elementet.

```
tl [1,2,3] ==> [2,3]
tl ["Hej"] ==> []
tl [] ger exekveringsfel (Empty).
```

`null` talar om ifall en lista är tom.

```
null [1,2,3] ==> false
null (tl ["Hej"]) ==> true
```

`::` (*cons*) skapar en ny listcell av ett *element* och en lista.

```
1::[2,3] ==> [1,2,3]          (:: är en infix operator)
```

`[1,2,3]` är egentligen `1::(2::(3::[]))`, vilket även kan skrivas `1::2::3::[]` (`::` är *högerassociativ*).

Polymorfa typer

`hd [1,2,3] ==> 1, hd ["Kalle","Anka"] ==> "Kalle"`
Vilken typ har `hd`? `int list->int`? `string list->string`?

`hd` har den *polymorfa* (mångformiga) typen

```
'a list -> 'a
```

Där `'a` är en *typvariabel*.
Alla typvariabler i ML har namn som börjar med apostrof.

`hd` kan användas i alla sammanhang där det behövs en typ som man kan få genom att ersätta `'a` med en annan typ, t.ex. `int` eller `string`. (Detta kallas en *instans* av `'a list -> 'a`).

`= : ''a*''a -> bool` (Dubbla apostrofer anger en *likhetstyp*.)

Listtyper är likhetstyper omm elementtypen är det!
Polymorfi är inte samma sak som överlagring (t.ex. hos +, <).

Specifikation av listoperationerna

```
hd l
Type: 'a list -> 'a
Pre: l <> []
Post: första elementet i l
tl l
Type: 'a list -> 'a list
Pre: l <> []
Post: l utan sitt första element.
null l                          Av tekniska skäl bör man
Type: 'a list -> bool           använda null (eller matchning)
Pre: (ingen)                    och inte direkt jämförelse
Post: l = []                    med tom lista.
x :: l
Type: 'a*'a list -> 'a list
Pre: (ingen)
Post: En lista som har x som första element och
      fortsätter som listan l.
```


Typen av tomma listan

Vilken typ har den tomma listan, `[]`?

`[]` kan vara en tom lista av heltal, en tom lista av strängar etc....

`tl [1]` ==> `[]`, `tl ["Kalle"]` ==> `[]`

`[]` måste alltså höra till både typen `int list` och `string list` (och alla andra sorters listor).

`[]` är ett *värde* som har den polymorfa typen `'a list`.

Definierade polymorfa funktioner

Polymorfi fungerar därför att den polymorfa funktionen (t.ex. `hd`) inte utför någon operation på *värden* av den obestämda typen.

Definierade funktioner blir också polymorfa om de har argument som de inte gör någon operation med:

```
(* swap(x,y)
   Type: 'a*'b -> 'b*'a *)
fun swap(x,y) = (y,x)

(* second l
   Type: 'a list -> 'a *)
fun second l = hd(tl l)

(* second0 l
   Type: int list -> int *) second0 blir inte polymorf.
fun second0 l = hd(tl l)+0
```


Strängar och teckenlistor

Strängar är en följd av tecken: `"Hej, hopp"`
Teckenlistor (`char list`):
`[#"H", #"e", #"j",#" ", #"h", #"o", #"p", #"p"]`
är också en följd av tecken – vad är skillnaden?

- Strängar och teckenlistor representeras (troligen) helt olika.
- Strängar kräver mindre minnesutrymme.
- Man har direkt åtkomst till delar av strängar utan att behöva "läsa förbi" den del av strängen som ligger före.
- Man kan matcha listor men inte strängar.
- Rekursion över listor är mer naturlig än över strängar.

Konverteringsfunktioner:

```
explode: string -> char list
implode: char list -> string
```


Räkna antalet element i listan

```
(* length l
   Type: 'a list->int
   Pre: (ingen)
   Post: Antal element i listan
   Ex: length [17,42] = 2
       length [] = 0 *)
(* Variant: längden av l *)
fun length l = if null l then
                 0
               else
                 1+length(tl l)

length [17,42]
==> if null [17,42] then 0 else 1+length(tl [17,42])
==> 1+length(tl [17,42])
==> 1+length [42]
==> 1+(if null [42] then 0 else 1+length(tl [42]))
==> 1+(1+length(tl [42]))
==> 1+(1+length [])
==> 1+(1+(if null [] then 0 else 1+length(tl [])))
==> 1+(1+0))
==> 2
```


Sätt samman två listor

```
(* append(x,y)
   Type: 'a list*'a list -> 'a list
   Pre: (ingen)
   Post: En lista med elementen i x följda av elementen i y.
   Ex: append([1,2],[3,4]) = [1,2,3,4]
       append([1,2],[]) = [1,2] *)
(* Variant: längden av x *)
fun append(x,y) = if null x then
                    y
                  else
                    hd x :: append(tl x,y)

append([1,2],[3,4])
==> if null [1,2] then [3,4] else hd [1,2] ::append(tl [1,2],[3,4])
==> hd [1,2]::append(tl [1,2],[3,4])
==> 1::append([2],[3,4])
==> 1::(if null [2] then [3,4] else hd [2]::append(tl [2],[3,4])
==> 1::(hd [2]::append(tl [2],[3,4])
==> 1::(2::append([],[3,4]))
==> 1::(2::(if null [] then [3,4] else hd [] ::append(tl [],[3,4])
==> 1::(2::[3,4])
==> 1::[2,3,4]
==> [1,2,3,4]
```


Mönstermatchning med listor

Listnotationen kan användas i matchning:

`[1,2,3]` matchar mönstret `[x,y,z]`.

`x` binds till `1`, `y` binds till `2` och `z` binds till `3`.

`::` är egentligen ingen funktion utan en (*datastruktur)konstruktor*.
Symbolen `::` *representerar* en listcell – och kan också användas i matchning.

`[1,2,3]` matchar mönstret `x::y`. `x` binds till `1`, `y` binds till `[2,3]`.

`[1,2,3]` matchar mönstret `x::y::z`. `x` binds till `1`, `y` binds till `2` och `z` binds till `[3]`.

Rekursion med mönstermatchning

```
(* append(x,y)
   Type: 'a list*'a list -> 'a list
   Pre: (ingen)
   Post: Elementen i x följda av de i y.
   Ex: append([1,2],[3,4]) = [1,2,3,4]
       append([1,2],[]) = [1,2] *)
(* Variant: längden av x *)
fun append([],y) = y
  | append(first::rest, y) = first::append(rest,y)
```

Obs. betydelsen av att namnge första argumentet i specifikationen – annars kan man inte skriva eftervillkoret på vettigt sätt!

```
append([1,2],[3,4])
==> 1::append([2],[3,4])
==> 1::(2::append([],[3,4]))
==> 1::(2::[3,4])
==> 1::[2,3,4]
==> [1,2,3,4]
```

Vänd på en lista

`append` använde vanlig (icke svans-) rekursion, vilket bevarade ordningen hos listelementen när listan byggs med `::`.
Svansrekursion ger omvänd ordning hos beräkningen, vilket är precis vad vi vill om vi skall vända ordningen på en lista.

```
(* rev l
   Type: 'a list -> 'a list
   Pre: (ingen)
   Post: En lista med elementen i l i omvänd ordning.
   Ex: rev [1,2,3,4]  = [4,3,2,1]
       rev [] = [] *)
fun rev l = revAux(l,[])
(* revAux(l,ack)
   Type: 'a list*'a list -> 'a list
   Pre: (ingen)
   Post: En lista med elementen i l i omvänd ordning följda av
         elementen i ack i rättvänd ordning.
   Ex: revAux([3,4],[2,1]) = [4,3,2,1] *)
(* Variant: längden av l *)
fun revAux([],ack) = ack
  | revAux(first::rest,ack) = revAux(rest,first::ack)
```


Vänd på en lista (forts.)

```
(* revAux l ack *)
fun revAux([],ack) = ack
  | revAux(first::rest,ack) =
      revAux(rest,first::ack)

(* rev l *)
fun rev l = revAux(l,[])

rev [1,2,3,4]
==> revAux([1,2,3,4],[])
==> revAux([2,3,4],[1])
==> revAux([3,4],[2,1])
==> revAux([4],[3,2,1])
==> revAux([],[4,3,2,1])
==> [4,3,2,1]
```

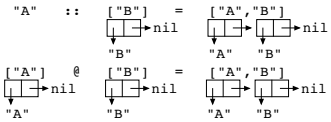

Fler inbyggda listfunktioner

`length`, `rev` och `append` är så vanliga att de är fördefinierade.
`append` heter som inbyggd funktion `@` och är en infix operator:

```
[1,2] @ [3,4] = [1,2,3,4]
```

OBS! Skilj på `::` (cons) och `@` (append)!

`::` skapar en ny listcell, medan `@` sätter ihop hela listor:



`"A" @ ["B"]` eller `["A"] :: ["B"]` ger typfel!

Komplexiteten hos append (`@`)

För att koppla ihop två listor måste den första listan *kopieras*.
Tidsåtgången för detta är O(*n*), där *n* är längden på första listan.

Antag att man försöker bygga en lista "framifrån" med hjälp av `@`:

`[] @ [1] ==> [1]`	0 rekursiva anrop av `@`
`[1] @ [2] ==> [1,2]`	1 rekursiva anrop av `@`
`[1,2] @ [3] ==> [1,2,3]`	2 rekursiva anrop av `@`
`[1,2,3] @ [4] ==> [1,2,3,4]`	3 rekursiva anrop av `@`

För att på detta sätt bygga en lista med *n* element krävs alltså
0+1+...+*n* = *n*(*n*+1)/2 = (n^2+*n*)/2 rekursiva anrop av `@`.
Tidsåtgången är *kvadratisk* – O(n^2)!

Bygg listan bakifrån med `::` istället! `1::(2::(3::(4::[])))`.
Tidsåtgången blir linjär – O(*n*).

Sökning i listor

Är ett bestämt värde ett element i en given lista?

```
(* member(x,l)
   Type: ''a*''a list -> bool
   Pre: (ingen)
   Post: true om x är ett element i l,false annars
   Ex: member(42,[17,42]) = true
       member(25,[17,42]) = false *)
(* Variant: längden av l *)
fun member(_,[]) = false
  | member(x,first::rest) = x = first orelse
                            member(x,rest)

member(42,[17,42])
==> 42 = 17 orelse member(42,[42])
==> 42 = 42 orelse member(42,[])
==> true
```

Notera att `member` är polymorf över *likhetstyper* (typvariabeln har dubbla apostrofer) – detta för att man gör en likhetsjämförelse.

Sökning i listor (2)

Hämta ett bestämt värde ur en tabell.

```
(* 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.
   Ex: 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)
```

Om `key` inte finns så avbryts exekveringen med ett fel. Vilket?
ML ger en varning när detta program läses in. Vilken?
Notera att man *inte* kan skriva: Varför?

```
fun lookup(key,(key,v)::rest) = v
  | lookup(key,_::rest) = lookup(key,rest)
```


Struktur hos program som arbetar med listor

Eftersom listor har godtycklig längd blir program som arbetar med listor i allmänhet rekursiva.

Oftast har rekursiva program över listor följande form:

```
(* f(...,l,...) *)
(* Variant: längden av l *)
fun f(...,[],...) = ...
  | f(...,first::rest,...) = ... f(...,rest,...) ...
```

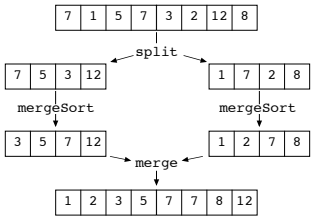
Denna typ av rekursion kallas *strukturell rekursion* – rekursion över *strukturen* hos listan.

Hanoi-programmet med listrepresentation

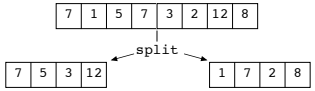
```
(* hanoi(n,from,via,to)
   Type: int*string*string*string->
         (string*string) list
   Pre: n>=0. from, via och to är olika strängar.
   Post: En lista av par som beskriver hur man
         skall flytta n brickor i Hanoi-spelet
         från pinne from till pinne to med hjälp
         av pinne via. Varje par anger att en
         bricka skall flyttas från den pinne som
         anges av första komponenten till den
         pinne som anges av andra komponenten.
   Ex.: hanoi(0,"A","B","C") = []
        hanoi(1,"A","B","C") = [("A","C")]
        hanoi(2,"A","B","C") = [("A","B"),("A","C"),("B","C")]
*)
(* Variant: n *)
fun hanoi(0,_,_,_) = []
  | hanoi(n,from,via,to) = hanoi(n-1,from,to,via) @
                           (from,to) :: hanoi(n-1,via,from,to)
```


Sortering

Princip för funktionen `mergeSort`.

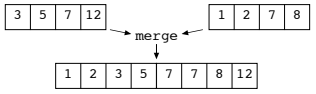


Uppdelning (split) av två listor




```
(* split l
   Type: 'a list -> 'a list*'a list
   Pre: (ingen)
   Post: (l1,l2) sådan att l1 och l2 innehåller elementen ifrån
         l i godtycklig ordning och längden av l1 och l2 som mest
         skiljer ett element.
   Ex.: split [7,1,5,7,3,2,12,8]  = ([7,5,3,12],[1,7,2,8] *)
(* Variant: length l *)
fun split [] = ([],[])
  | split (first::rest) = let
                            val (l2,l1) = split rest
                          in
                            (first::l1, l2)
                          end
```


Sammanvävning (merge) av två listor




```
(* merge(l1,l2)
   Type: int list*int list -> int list
   Pre: l1 och l2 är sorterade i stigande ordning.
   Post: En lista med alla element i l1 och l2 i stigande ordning.
   Ex.: merge([3,5,7,12],[1,2,7,8]) = [1,2,3,5,7,7,8,12] *)
(* Variant: (length l1)+(length l2) *)
fun merge([],l2) = l2
  | merge(l1,[]) = l1
  | merge(l1,l2) = if hd l1 < hd l2 then
                     hd l1 :: merge(tl l1, l2)
                   else
                     hd l2 :: merge(l1,tl l2)
```


mergeSort

```
(* mergeSort l
   Type: int list -> int list
   Pre: (ingen)
   Post: En lista med alla element i l i stigande ordning.
   Ex.: mergeSort [7,1,5,7,3,2,12,8] = [1,2,3,5,7,7,8,12] *)
(* Variant: length l *)
fun mergeSort [] = []
  | mergeSort [x] = [x]
  | mergeSort l = let
                    val (l1,l2) = split l
                  in
                    merge(mergeSort l1, mergeSort l2)
                  end
```

Varför krävs två basfall – tom lista och enelementslista?
Varför är inte `mergeSort` polymorf?
Komplexiteten hos `mergeSort` är O(*n* log *n*), vilket är typiskt för sorteringsalgoritmer.
(*n* är alltså storleken på problemet – i detta fall listans längd.)