"Referentiell transparens" igen

Utan sidoeffekter kan ett ML-program aldrig ändra någonting – bara skapa nya objekt. Detta är en förutsättning för att man skall kunna byta "lika mot lika".

```
- val x = [1,2,3,4];
> val x = [1, 2, 3, 4] : int list
- rev x;
> val it = [4, 3, 2, 1] : int list
- x;
> val it = [1, 2, 3, 4] : int list
```

`x` är oförändrad efter anropet av `rev x`!

```
- val y = 1;
> val y = 1 : int
- fun f(x) = x+y;
> val f = fn : int -> int
- y;
> val it = 1 : int
- f(2);
> val it = 3 : int
- val y = 2;
> val y = 2 : int
- y;
> val it = 2 : int
- f(2);
> val it = 3 : int
```

Den andra deklarationen `val y` ser kanske ut att förändra `y`, men det gör den inte. Den gör en *ny* definition av `y` som skuggar den gamla. Den gamla finns kvar som man kan se av anropen till `f`!

Modifierbara celler

ML har möjlighet till "äkta" modifiering av data (sidoeffekter).

```
ref : 'a -> 'a ref
!   : 'a ref -> 'a
:=  : 'a ref * 'a -> unit
```

`ref` och `!` beter sig nästan som om de definierats genom:

```
datatype 'a ref = ref of 'a
fun ! (ref x) = x
```

`ref` skiljer sig från andra datatyper genom att man kan *ändra* det taggade objektet! Den infixa operatorn `:=` ändrar *innehållet* i sitt första argument till sitt andra argument.

```
- val y = ref 1;
> val y = ref 1 : int ref
- fun f(x) = x + !y;        (inte x +!y)
> val f = fn : int -> int
- !y;
> val it = 1 : int
- f(2);
> val it = 3 : int
- y := 2;
> val it = () : unit
- !y;
> val it = 2 : int
- f(2);
> val it = 4 : int
```

`y` är här en *referens* till ett heltal – kallas också cell (box, låda) som innehåller ett heltal.

Celler är "första klassens objekt" i ML och kan t.ex. vara delar av andra objekt:

```
- [y, ref 2];
> val it = [ref 1, ref 2] : int ref list
```

Likhet och olikhet hos celler

```
- val y = ref 1;
> val y = ref 1 : int ref
- y;
> val it = ref 1 : int ref
- !y;
> val it = 1 : int
- val y' = y;
> val y' = ref 1 : int ref
- y := 2;
> val it = () : unit
- y;
> val it = ref 2 : int ref
- !y;
> val it = 2 : int
- val z = ref 2;
> val z = ref 2 : int ref
```

Att cellen `y` tilldelas ett nytt värde (`2`) ändrar inte identiteten hos cellen – det är fortfarande samma cell.

```
- y = y';
> val it = true : bool
```

En nyskapad cell (`z`) med samma innehåll är däremot en annan cell!

```
- y = z;
> val it = false : bool
```

Eftersom likhet mellan `ref`-objekt är likhet mellan celler är `'a ref` alltid en likhetstyp även om innehållstypen (`'a`) inte är det! Detta är annorlunda mot hur konstruerade datatyper
normalt fungerar.

```
- val f = ref (op +);
> val f = ref fn : (int * int -> int) ref
- val g = ref (op +);
> val g = ref fn : (int * int -> int) ref
- f=g;
> val it = false : bool
```

ML-systemets "lager" (store)

Sedan tidigare vet vi att:

- Definitioner (`val`, `fun`) binder identifierare till värden.
- Bindningsomgivningar är en samling bindningar.
- Det finns flera bindningsomgivningar – de kan skapas och försvinna.

Nu tillkommer ML's "lager" (store) som är en samling celler.

- Cellen innehåller ett värde
- Lagret är en samling celler
- Lagret kan *ändras* genom att cellernas innehåll ändras.
- Det finns bara *ett* lager och det försvinner aldrig.

Bindningsomgivning	Lager
`val y = ref 1`	
`y: r1`	`r1:` 1
`val y' = y`	
`y: r1` `y': r1`	`r1:` 1
`y := 2`	
`y: r1` `y': r1`	`r1:` 2
`val z = ref 2`	
`y: r1` `y': r1`	`r1:` 2
`z: r2`	`r2:` 2

Nu är `y=y'`, men `y≠z`.

Pekare

Man kan låta innehållet i en cell själv vara en cell.

Bindningsomgivning	Lager

```
val x = ref 1
val y = ref 2
```

```
   x: r1                    r1:   1
   y: r2                    r2:          2
```

```
val z = ref x
val u = ref (!x)
```

`z` får typ `int ref ref` – en *pekare* till en annan cell (x)
u blir en ny cell med samma innehåll som x.

```
   x: r1                    r1:   1
   y: r2                    r2:          2
   z: r3                    r3:   
   u: r4                    r4:          1
```

```
!z := 3
```

```
   x: r1                    r1:   3
   y: r2                    r2:          2
   z: r3                    r3:   
   u: r4                    r4:          1
```

```
z := y
```

```
   x: r1                    r1:   3
   y: r2                    r2:          2
   z: r3                    r3:   
   u: r4                    r4:          1
```

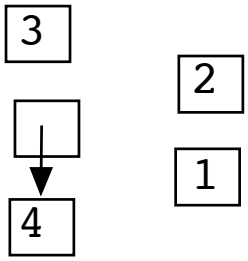
```
z := ref 4
```

```
   x: r1                    r1:   3
   y: r2                    r2:          2
   z: r3                    r3:   
   u: r4                    r4:          1
                            r5:   4
```

Skräpsamling

```
x: r1
y: r2
z: r3
u: r4
```

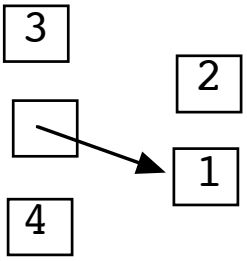
```
r1:
r2:
r3:
r4:
r5:
```




```
z := u
```

```
x: r1
y: r2
z: r3
u: r4
```

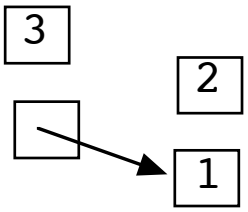
```
r1:
r2:
r3:
r4:
r5:
```



Nu finns ingen referens till r5 någonstans. Cellen r5 har blivit *skräp* som kan tas bort automatiskt. Detta kallas skräpsamling (garbage collection).

```
x: r1
y: r2
z: r3
u: r4
```

```
r1:
r2:
r3:
r4:
```



OBS! Alla sorters data i ML kan bli skräp – inte bara celler – men fenomenet märks kanske tydligast med celler.

Tack vare skräpsamlingen fungerar det att ML-program arbetar med att ständigt konstruera nya objekt utan att någonsin ta bort gamla.

Imperativ programmeringsstil

I imperativ programmeringsstil arbetar man med att

- *uppdatera* innehållet i celler snarare än binda identifierare
- utföra beräkningar i steg (*satser*), snarare än som funktionsuttryck
- utföra upprepningar med *loopar* snarare än rekursion.

```
(* absSqrt(x)  : real -> real
   PRE: (inget)
   POST: Kvadratroten ur absolutbeloppet
         av x
*)
fun absSqrt(x) =
  if x < 0.0 then
    Math.sqrt(~x)
  else
    Math.sqrt(x)

fun absSqrt'(x) =
  let
    val x' = ref x
  in
    if !x' < 0.0 then
      x' := ~ (!x')
    else
      () ;
    Math.sqrt(!x')
  end
```

Loopar

En loop *upprepar* ett uttryck (sats).

`while` e_1 `do` e_2

Uttrycket e_2 beräknas upprepade gånger så länge uttrycket e_1 beräknas till `true`. Värdet av `while` är alltid `()`.

Utan sidoeffekter är detta meningslöst.

`while` har följande beräkningsregler:

`while` e_1 `do` e_2 `~>` `()` Om e_1 `~>` `false`

`while` e_1 `do` e_2 Om e_1 `~>` `true`
`~>` e_2`;` `while` e_1 `do` e_2

```
let
  val i = ref 1
in
  while !i<=10 do
    (print "Hej igen!\n";
     i := !i + 1)
end
```

Varje varv i loopen skrivs "Hej, igen" ut och innehållet i cellen i ökas med 1. När innehållet överstiger 10 avbryts loopen.

`while` är f.ö. syntaktiskt socker.
Istället för `while` e_1 `do` e_2 kan man lika gärna skriva:

```
let
  fun aux() = if e1 then (e2;aux()) else ()
in
  aux()
end
```

Imperativ fakultetsfunktion

Fakultetsfunktionen:

```
(* fact(x)  : int -> int
   PRE: x >= 0
   POST: 1*2*....*x
*)
fun fact(x) =
  let
    val n = ref x
    val p = ref 1
  in
    while !n > 0 do
      (p := !p * !n;
       n := !n - 1);
    !p
  end
```

fact(3)
~> let **val n = ref 3 val p = ref 1** in .. end

(n och p binds till var sin cell med innehåll 3 resp 1)

~> while **!n>0** do (p:= !p* !n;n:= !n-1);!p

(**!n**>0 ~> **3>0** ~> true)

~> (p:= **!p*** **!n;**n:= !n-1); while...
~> (p:=**1*3;**n:= !n-1); while...
~> (**p:=3;**n:= !n-1); while...

(innehållet i p ändras till 3)

~> (n:= **!n**-1); while...
~> (n:=**3-1**); while...
~> (**n:=2**); while...

(innehållet i n ändras till 2)

~> while !n>0 do (p:= !p* !n;n:= !n-1);!p

Imperativ fakultetsfunktion (forts.)

(Cellerna som n och p är bundna till innehåller nu talen 2 resp 3)

```
~> while !n>0 do (p:= !p* !n;n:= !n-1);!p
(!n>0 ~> 2>0 ~> true)
~> (p:= !p* !n;n:= !n-1); while...
~> (p:=3*2;n:= !n-1); while...
~> (p:=6;n:= !n-1); while...
```

(innehållet i p ändras till 6)

```
~> (n:= !n-1); while...
~> (n:=2-1); while...
~> (n:=1); while...
```

(innehållet i n ändras till 1)

```
~> while !n>0 do (p:= !p* !n;n:= !n-1);!p
(!n>0 ~> 1>0 ~> true)
~> (p:= !p* !n;n:= !n-1); while...
~> (p:=6*1;n:= !n-1); while...
~> (p:=6;n:= !n-1); while...
```

(innehållet i p ändras till 6)

```
~> (n:= !n-1); while...
~> (n:=1-1); while...
~> (n:=0); while...
```

(innehållet i n ändras till 0)

```
~> while !n>0 do (p:= !p* !n;n:= !n-1);!p
(!n>0 ~> 0>0 ~> false)
~> !p
~> 6
```

Längden på listor:

```
(* length(x)  : 'a list -> int
   PRE: (inget)
   POST: Längden på listan x
*)
fun length(x) =
  let
    val l = ref x
    val n = ref 0
  in
    while not (null(!l)) do
      (n := !n + 1;
       l := tl(!l));
    !n
  end
```

Antal stora element:

```
(* countGreater(x,g)  : 'a list*int -> int
   PRE: (inget)
   POST: Antal element i listan x som är
         större än g
*)
fun countGreater(x,g) =
  let
    val l = ref x
    val n = ref 0
  in
    while not (null(!l)) do
      (if hd(!l) > g then
         n := !n + 1
       else
         ();
       l := tl(!l));
    !n
  end
```

Imperativa datastrukturer

`counter` är en abstrakt datatyp som innehåller en räknare. Räknaren kan räknas upp och avläsas med sidoeffekter.

```
abstype counter = Counter of int ref
with
  fun makeCounter() = Counter(ref 0)
  fun incCounter(Counter x) = x:= !x+1
  fun readCounter(Counter x) = !x
end

- val a = makeCounter();
> val a = <counter> : counter
- incCounter(a);
> val it = () : unit
- incCounter(a);
> val it = () : unit
- readCounter(a);
> val it = 2 : int
- incCounter(a);
> val it = () : unit
- readCounter(a);
> val it = 3 : int
```

Användning av `counter`

Räkna antalet additioner som utförs i följande uttryck.

```
fun test2(v) =
    let
        fun x(a) = a + a;
        fun y(a) = x(a) + x(a);
    in
        y(v) + y(v)
    end;
```

Definiera om `test2` med användning av `incCounter`:

```
- val c = makeCounter();
> val c = <counter> : counter
- fun test2'(v) =
  let
    fun x(a) = (incCounter(c); a + a);
    fun y(a) = (incCounter(c); x(a) + x(a));
  in
    (incCounter(c); y(v) + y(v))
  end;
> val test2' = fn : int -> int
- test2'(1);
> val it = 8 : int
- readCounter(c);
> val it = 7 : int
```

Loopar och iteration:

Iteration (svansrekursion) och loopar med tilldelning är i princip samma sak – man kan enkelt skissa en översättning mellan dem.

```
fun f(x) =
  if test(x) then
    f(dosomething(x))
  else
    conclude(x)
```

Den svansrekursive funktionen `f` kommer att beräkna

`conclude(dosomething(dosomething(...x)))`

där antalet `dosomething` bestäms av antalet rekursioner innan testet misslyckas.

```
fun f'(x) =
  let
    val x' = ref x
  in
    while test(!x') do
      x' := dosomething(!x');
    conclude(!x')
  end
```

Den imperativa funktionen `f'` kommer att sätta innehållet i `x'` till `dosomething` av innehållet i `x'` ett antal gånger som bestäms av antalet loopvarv innan testet misslyckas. Därefter anropas `conclude` på resultatet. Även `f'` kommer alltså att beräkna.

`conclude(dosomething(dosomething(...x)))`

Eftersom svansrekursion är ett speciallfall av allmän rekursion följer att rekursion är en mer kraftfull metod att konstruera program än loopar.

Imperativ och svansrekursiv fakultetsfunktion

Svansrekursiv:

```
local
  fun factaux(n,p) =
    if n > 0 then
      factaux(n-1,n*p)
    else
      p
in
  fun fact(x) = factaux(x,1)
end;
```

Imperativ:

```
fun fact(x) =
  let
    val n = ref x
    val p = ref 1
  in
    while !n > 0 do
      (p := !p * !n;
       n := !n - 1);
    !p
  end
```

Tänk på att ordningen av satserna spelar roll! Man kan inte skriva loopkroppen som:

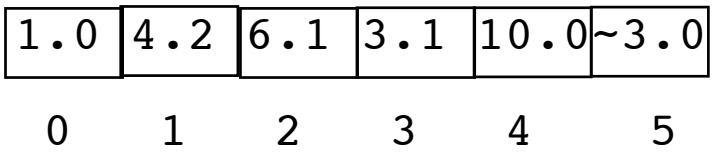
```
      (n := !n - 1;
       p := !p * !n)
```

Arrayer

Man har ibland behov av en datastruktur där man kan komma åt eller ändra ett element i konstant tid. Detta är relevant eftersom nästan alla datorer har ett arbetsminne med ett antal sekventiellt ordnade celler som alla går att komma åt eller ändra i konstant tid.

Listor, träd o.dyl. rekursiva datastrukturer duger i så fall inte eftersom man måste söka sig fram till det element man vill komma åt. Även om sökningen kan göras i O(log n) tid så kommer tidsåtgången att öka med storleken hos datastrukturen.

De flesta programspråk har en datastruktur som direkt motsvarar strukturen hos minnet – *arrayen* (vektor, fält).



En flyttalsarray med 6 element och deras *index*.

Till skillnad från en lista som består av ett antal `::`-konstruktorer är en array *ett* objekt – att komma åt alla element går alltså lika fort.

Arrayer i ML definieras i biblioteket `Array` och liknar en abstrakt datatyp. Det är dock en likhetstyp. (Men observera att liksom hos `ref` så kan två arrayer vara olika även om innehållet är lika!)

Arrayen ovan har typen `real Array.array`.

Arrayer skiljer sig från `ref` i princip genom att `ref` är *en* cell, medan arrayer normalt har *flera* celler.

Funktioner i biblioteket `Array` (urval)

(På denna sida antar vi att `open Array` gjorts, för enkelhets skull)

```
fromList(x): 'a list -> 'a array
POST: En array av samma längd som listan
      där cellerna i tur och ordning
      innehåller elementenn i listan
SIDE-EFFECTS: En ny array skapas.

array(n,x): int*'a  -> 'a array
PRE:  n >= 0
POST: En array av längd n
      där cellerna innehåller x
SIDE-EFFECTS: En ny array skapas.

length(a): 'a array  -> int
POST: Längden hos arrayen a

sub(a,i): 'a array*int  -> 'a
PRE:  0 <= i < length(a)
POST: Innehållet hos cellen med index i
      inom arrayen A.

update(a,i,x): 'a array*int*'a  -> unit
PRE:  0 <= i < length(a)
SIDE-EFFECT: I arrayen A sätts innehållet
             i cellen med index i till x.

foldr f e A
   ('a*'b->'b)->'b->'a array->'b
POST: a1 f (a2 f (... f (an f e)))
      om cellerna i arrayen A
      innehåller a1,a2,...,an
```

Jämför med `foldr` för listor!

Användning av arrayfunktioner

```
(* toList(x): 'a Array.array -> 'a list
   POST: En lista av elementen i arrayen a.
*)
fun toList a = Array.foldr (op ::) [] a;
```

`toList` är bra att ha för att kunna se hur en array ser ut!

```
- val a = Array.fromList [10,20,30,40];
> val a = <array> : int array
- toList(a);
> val it = [10, 20, 30, 40] : int list
- Array.length(a);
> val it = 4 : int
- Array.sub(a,3);
> val it = 40 : int
- Array.update(a,2,42);
> val it = () : unit
- toList(a);
> val it = [10, 20, 42, 40] : int list
- val b = Array.array(3,0);
> val b = <array> : int array
- toList(b);
> val it = [0, 0, 0] : int list
- a=b;
> val it = false : bool
```

Summera elementen i en flyttalsarray

Funktionellt:

```
local
  fun sumArrayAux(a,i) =
    if i >= Array.length(a) then
      0.0
    else
      Array.sub(a,i) + sumArrayAux(a,i+1)
in
  fun sumArray(a) = sumArrayAux(a,0)
end
```

Imperativt:

```
fun sumArray'(a) =
  let
    val sum = ref 0.0
    val i = ref 0
  in
    while !i < Array.length(a) do
      (sum := !sum + Array.sub(a,!i);
       i := !i + 1);
    !sum
  end
```

Bekvämt:

```
fun sumArray''(a) = Array.foldl (op +) 0.0 a
```

```
- val a = Array.fromList [1.0, 2.0, 3.0];
> val a = <array> : real array
- sumArray a;
> val it = 6.0 : real
```

Fibonaccital

```
(* fibonArray(n) : int -> int Array.array
   PRE: n >= 0
   POST: En array av längd n innehållande
         de n första fibonaccitalen
   SIDE-EFFECTS: En ny array skapas
*)
local
  (* fibonArrayAux(a,i) :
        int Array.array*int->int Array.array
     PRE: 2 <= i
     POST: Arrayen a
     SIDE-EFFECTS: Arraycellen med index i
                   sätts till summan av
                   innehållet i de två
                   föregående cellerna.
  *)
  fun fibonArrayAux(a,i) =
    if i >= Array.length(a) then
      a
    else
      (Array.update(a,i,Array.sub(a,i-1)+
                        Array.sub(a,i-2));
       fibonArrayAux(a,i+1))
in
  fun fibonArray(n) =
     fibonArrayAux(Array.array(n,1), 2)
end
```

`fibonArray` skapar en array med fibonaccital. Arrayen skapas med alla celler satta till 1 (vilket är de korrekta 1:a och 2:a fibonaccitalen). Därefter beräknas återstående fibonaccitalet genom att de två föregående adderas.

```
- fibonArray(10);
> val it = <array> : int array
- toList(it);
> val it = [1, 1, 2, 3, 5, 8, 13, 21, 34,
55] : int list
```