"Referentiell transparens"

Hittills har jag beskrivit körningen av ML-program genom att uttryck ersätts med sina värden:

```
fun fact(0) = 1
  | fact(n) = n*fact(n-1)
```

```
fact(3)
~> 3*fact(3-1) ~> 3*fact(2)
~> 3*(2*fact(2-1)) ~> 3*(2*fact(1))
~> 3*(2*(1*fact(1-1))) ~> 3*(2*(1*fact(0)))
~> 3*(2*(1*1)) ~> 3*(2*1) ~> 3*2 ~> 6
```

Det enda "effekt" evalueringen av ett uttryck har är alltså att beräkna dess värde. (Jag bortser här från resursförbrukning – tid, minne...). Man kan alltså fritt ersätta ett uttryck med ett annat uttryck som har samma värde, t.ex. kan

`f(x)*g(y)` och `g(y)*f(x)`

fritt bytas mot varandra i ett ML-program eftersom multiplikation är kommutativ. Det spelar ingen roll hur funktionerna `f` och `g` ser ut. Evalueringsordningen spelar alltså heller ingen roll (observera dock att man alltid anropar en funktion med fullständigt evaluerade uttryck – ivrig evaluering).

Detta gör ML-program lättare att förstå och ger också möjlighet för ML-systemet att "förbättra" programmet.

`f(x)^f(x)` kan t.ex. ersättas av

```
val fx = f(x) in fx^fx end
```

...vilket lönar sig resursmässigt om beräkningen av `f(x)` tar lång tid eller har ett stort objekt som värde.

Att man kan byta "lika mot lika" kallas *referentiell transparens* och är det normala hos funktionella språk.

Inmatning: problem

Hittills har vi anropad ML-funktioner med hjälp av ML-systemet. ML-systemet har hanterat inläsning av indata och utskrift av resultat till/från ett terminalfönster. I praktiken räcker inte det: man vill kunna styra inläsning och utskrift själv och läsa/skriva även på filer.

Låt oss anta att det finns en funktion `readLine`:

```
readLine x : unit -> string
SIDE-EFFECTS: En rad läses in från
              terminalfönstret
POST: en sträng som innehåller den
      rad som lästs in. (x ignoreras)
EXAMPLE: readLine() = "Hej\n" om man
         skriver in raden "Hej\n" på
         tangentbordet.
```

Vad blir `readLine(x)^readLine(x)`? Låt oss anta att användaren först skriver raden "Hej", sedan "hopp". (Varför slutar strängarna med `\n`?)

```
readLine()^readLine()
~>"Hej\n"^readline()
~>"Hej\n"^"hopp\n"
~>"Hej\nhopp\n"
```

Observera att `readLine() ≠ readLine()`!! Det beror på att `readLine` inte bara lämnar tillbaka ett värde utan också läser en rad från terminalen. En *ny* rad läses en gång vid varje anrop. *Antalet* anrop blir väsentligt.

```
let val r = readLine() in r^r end
~> val r = "Hej\n" in r^r end
~> "Hej\n"^"Hej\n"
~> "Hej\nHej\n"
```

Anropet av `readLine` gör annat än bara beräknar värdet. Värdet bestäms av faktorer som kan ändras från anrop till anrop. `readLine` har en *sidoeffekt*.

Sekvensering

```
x ; y : 'a*'b->'b
POST: Värdet av y

infix 0 ;
```

Uttrycket `x;y` har som värde värdet av uttrycket `y`. Utan sidoeffekter vore `;` onödig eftersom `x;y = y` , så alla förekomster av `x;y` kunde bytas ut mot `y`. Syftet med att skriva `x;y` är just möjligheten att det annars meningslösa uttrycket till vänster har en sidoeffekt.

(Egentligen är `;` ingen operator utan ett "nyckelord", men det fungerar ungefär som en operator.)

Eftersom förekomster av sidoeffekter gör att lika inte kan bytas mot lika blir *evalueringsordningen* viktig. I ML beräknas uttryck från vänster till höger. Man kan alltså vara säker på att i `x;y` så beräknas `x` före `y`.

Grupperingar med parentes ändrar inte detta – det vänstraste *hela deluttrycket* beräknas först.

```
fun square(x) = x*x

square(2*3)*(4+5)
~> square(6)*(4+5)
~> (6*6)*(4+5)
~> 36*(4+5)
~> 36*9
~> 324
```

Exekveringsordningen är viktig att hålla reda på när man programmerar med sidoeffekter.

Utmatning: samma problem

Funktioner i ML för utmatning ger också problem. Låt oss använda funktionen `print`:

```
print x : string -> ()
SIDE-EFFECTS: Strängen x skrivs ut på
              terminalfönstret.
POST: ()
EXAMPLE: print("Hej\n") = () och
         skriver samtidigt raden "Hej\n"
         på terminalfönstret.
```

Vad blir `print("Hej\n");print("Hej\n")`?

```
print("Hej\n");print("Hej\n")
~> ();print("Hej\n")
~> ();()
~> ()
```

Samtidigt skrivs raden `"Hej\n"` ut två gånger. Observera att `print("Hej\n") = print("Hej\n")`, men det hjälper inte! Det beror på att `print` inte bara lämnar tillbaka ett värde utan också skriver en rad på terminalen. En *ny* rad skrivs en gång vid varje anrop. *Antalet* anrop blir väsentligt.

```
let val p = print("Hej\n") in p;p end
~> val p = () in p;p end
~> ();()
~> ()
```

Samtidigt skrivs raden `"Hej\n"` ut *en* gång. Även `print` har sidoeffekt!

In- och utmatning i ML – `TextIO`

Det *går* att utföra in/utmating i ett funktionellt språk på ett sätt som i stort sett helt undviker sidoeffekter – speciellt om språket har lat evaluering. Så har man tyvärr inte gjort i ML, utan all I/O i ML är baserad på sidoeffekter.

För I/O finns olika bibliotek i ML, bl.a. biblioteket `TextIO` för in- och utmatning av text. För att slippa skriva `TextIO.` före alla namn på funktioner och värden i detta bibliotek kan man använda deklarationen

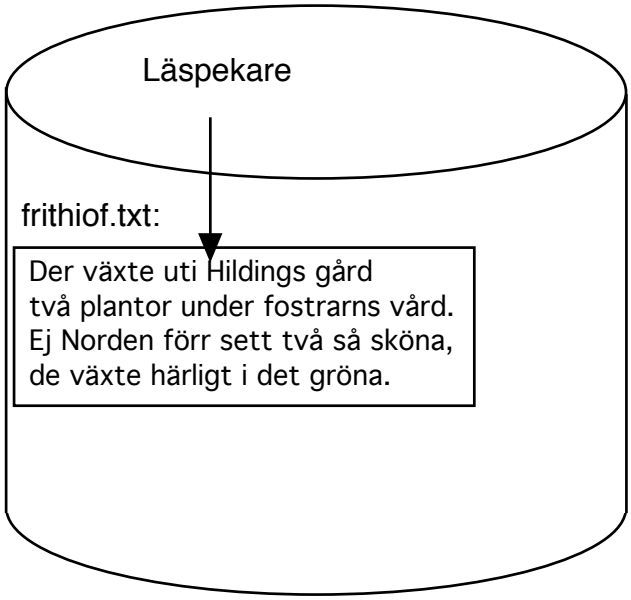
```
open TextIO;
```

i sitt program. Den gör att ML automatiskt söker i biblioteket `TextIO` efter namn den inte känner igen annars. (Detta fungerar med alla bibliotek, även t.ex. `String` och `List`.) Man kan använda flera `open`-deklarationer. Jag förutsätter i fortsättningen att man använt deklarationen `open TextIO`.

(Obs! `open` är ingen ersättning för `load`.)

In- och utmatning i ML – strömmar

Vid in/utmatning krävs en omfattande administration, bl.a. skall filer lokaliseras och man behöver hålla reda på hur långt i den aktuella filen man har läst/skrivit.



För detta ändamål har ML i `TextIO` två abstrakta datatyper `instream` och `outstream` kallade *strömmar*. All information rörande adminstrationen av in/utmatning finns i strömmarna.

För in/utmatning till terminalfönster finns i `TextIO` två fördefinierade strömmar `stdIn` av typ `instream` och `stdOut` av typ `outstream`.

```
openIn(s) : string -> instream
PRE: s skall vara namnet på en fil
     som finns.
POST: En instream kopplad till filen s.
EXAMPLE : openIn("frithiof.txt")
SIDE-EFFECTS: Olika anrop till openIn
              ger olika strömmar.

openOut(s) : string -> outstream
PRE: s skall vara namnet på en fil
     som går att skriva/skapa.
POST: En outstream kopplad till filen s.
SIDE-EFFECTS: Filen s skapas om den inte
              finns. Olika anrop till
              openOut ger olika strömmar.
```

Inmatning – ett urval funktioner

```
inputLine(is) : instream -> string
POST: Nästa rad från strömmen is avslutad
      med #"\n", eller tom sträng om
      inget kunde läsas (t.ex. EOF)
SIDE-EFFECTS: Läspekaren för is flyttas
              fram.

input1(is) : instream -> char option
POST: SOME c om nästa tecken i strömmen
      är c, NONE om inget tecken kunde
      läsas (t.ex. EOF).
SIDE-EFFECTS: Läspekaren för is flyttas
              fram.

inputAll(is) : instream -> string
POST: Resterande text i strömmen is
      till filens slut.
SIDE-EFFECTS: Läspekaren för is flyttas
              fram.

endOfStream(is) : instream -> bool
POST: true om läsningen av strömmen is
      har nått till slutet av filen,
      false annars.
SIDE-EFFECTS: inga.

closeIn(is) : instream -> unit
POST: ().
SIDE-EFFECTS: Avslutar användningen av
              strömmen is.
```

Exempelfunktionen `readline` kan definieras så här:

```
fun readline() = inputLine(stdIn);
```

Inmatning: exempel

Om vi har en fil `testin.txt`, med innehåll:

```
Detta är första raden...
...och detta är andra.
En rad mitt i...
...och sista
```

Så kan vi göra följande:

```
- val x = openIn("testin.txt");
> val x = <instream> : instream
- inputLine(x);
> val it =
 "Detta är första raden...\n" : string
- input1(x);
> val it = SOME #"." : char option
- input1(x);
> val it = SOME #"." : char option
- inputLine(x);
> val it = ".och detta är andra.\n" : string
- endOfStream(x);
> val it = false : bool
- inputAll(x);
> val it =
 "En rad mitt i...\n...och sista\n" : string
- endOfStream(x);
> val it = true : bool
- input1(x);
> val it = NONE : char option
- inputLine(x);
> val it = "" : string
- closeIn(x);
> val it = () : unit
-
```

Utmatning – ett urval funktioner

```
output(os,s) : outstream*string -> unit
POST: ()
SIDE-EFFECTS: Strängen s skrivs ut på
              strömmen os.

output1(os,c) : outstream*char -> unit
POST: ()
SIDE-EFFECTS: Tecknet c skrivs ut på
              strömmen os.

flushOut(os) : outstream -> ()
POST: ().
SIDE-EFFECTS: Skriver ut bufferten för
              strömmen os.

closeOut(os) : outstream -> unit
POST: ().
SIDE-EFFECTS: Avslutar användningen av
              strömmen os.
```

Exempelfunktionen `print` finns fördefinierad så här:

```
fun print(s) = (output(stdOut,s);
                flushOut(stdOut);
```

Utmatning: exempel

Så kan vi göra följande:

```
- val x = openOut("testut.txt");
> val x = <outstream> : outstream
- output(x,"Hej");
> val it = () : unit
- output1(x,#",");
> val it = () : unit
- output(x," Hopp\n");
> val it = () : unit
- output(x,"I det gröna\n");
> val it = () : unit
- closeOut(x);
> val it = () : unit
-
```

Nu finns en fil `testut.txt`, med innehåll:

```
Hej, Hopp
I det gröna
```

In/utmatning mot terminalfönster

Exempel på in/utmatning mot terminalen. Inmatad text är satt med **fetstil**. Se upp med buffring av in- och utdata!:

```
- inputLine(stdIn);
abcd
> val it = "abcd\n" : string
- output(stdOut,"Den röda räven rev en annan
räv\n");
Den röda räven rev en annan räv
> val it = () : unit
- output(stdOut,"Här saknas radslut:");
Här saknas radslut:> val it = () : unit
- (output(stdOut,"Här buffras utdata!\n");
inputLine(stdIn));
Nu skriver jag in en rad
Här buffras utdata!
> val it = "Nu skriver jag in en rad\n" :
string
- (output(stdOut,"Här töms bufferten
först\n"); flushOut(stdOut);
inputLine(stdIn));
Här töms bufferten först
Nu skriver jag in en ny rad
> val it =
  "Nu skriver jag in en ny rad\n" : string
- (print("Det går lika bra med print\n");
inputLine(stdIn));
Det går lika bra med print
Nu skriver jag in en 3:e rad
> val it =
  "Nu skriver jag in en 3:e rad\n" : string
- input1(stdIn);
Nu skriver jag en rad igen
> val it = SOME #"N" : char option
- 2+2;
! Toplevel input:
! u skriver jag en rad igen
! ^
! Unbound value identifier: u
```

Tillägg till befintliga filer

`openOut` skapar en ny fil eller skriver över en befintlig fil.
Man kan också vilja lägga till ny information i slutet av en fil.

```
openAppend(s) : string -> outstream
PRE: s skall vara namnet på en fil
     som går att skriva/skapa.
POST: En outstream kopplad till filen s.
SIDE-EFFECTS: Filen s skapas om den inte
              finns. Skrivningen börjar
              i slutet av filen.
              Olika anrop till openAppend
              ger olika strömmar.
```

Om vi har en fil `testut.txt`, med innehåll:

```
Hej, Hopp
I det gröna
```

och gör:

```
- val x = openAppend("testut.txt");
> val x = <outstream> : outstream
- output(x,"sade Frithiof\n");
> val it = () : unit
- closeOut(x);
> val it = () : unit
```

så kommer `testut.txt` att innehålla:

```
Hej, Hopp
I det gröna
sade Frithiof
```

In- och utmatning av annat än text

Ofta vill man läsa och skriva annan information än text – t.ex. tal. I så fall måste man översätta mellan talen och en strängrepresentation av dem.

```
(* intToString(i) : int -> string
   POST: En textrepresentation av talet i.
   SIDE-EFFECTS: inga.
fun intToString(i) =
  let
    (* iTSAux(i) : int -> string
       PRE: i > 0
       POST: En textrepresentation av i.
       SIDE-EFFECTS: inga.
       INDUCTION VARIABLE: i
       WELL-FOUNDED RELATION: < *)
    fun iTSAux(0) = ""
      | iTSAux(n) =
          iTSAux(n div 10) ^
          Char.toString(chr(ord(#"0")+
                            (n mod 10)))
  in
    if i = 0 then
      "0"
    else if i < 0 then
      "~" ^ iTSAux(~i)
    else
      iTSAux(i)
  end



- print(intToString(2) ^ " gånger " ^
intToString(3) ^ " är " ^ intToString(2*3) ^
"\n");
2 gånger 3 är 6
> val it = () : unit
```

Inmatning av tal är svårare

Försöker man konvertera en sträng till ett tal så kan det hända att strängen inte representerar ett korrekt tal, t.ex. "a12b".

```
(* intFromString(s) : string -> int option
   POST: SOME i om strängen är en
         representation av talet i,
         annars NONE
   SIDE-EFFECTS: inga. *)
fun intFromString(s) =
  let
    (* iFSAux(n, acc): int*int -> int option
       PRE: n >= 0 och n <= size(s)
       POST: SOME i+acc*(10**(size(s)-n)),
             om s från pos. n till slutet
             representerar talet i,
             annars NONE
       SIDE-EFFECTS: inga.
       INDUCTION-VARIABLE: n
       WELL-FOUNDED ORDERING:
         < på size(s)-n *)

    fun iFSAux(n, acc) =
      if n = size(s) then
        SOME acc
      else
        let
          val digit =
              ord(String.sub(s,n))-ord(#"0")
        in
          if digit < 0 orelse digit > 9 then
            NONE
          else
            iFSAux(n+1, acc*10+digit)
        end
  in
```

Inmatning av tal är svårare (forts.)

```
    if size(s) > 0 andalso
       String.sub(s,0) = #"~" then
      if size(s) = 1 then
        NONE
      else
        case iFSAux(1, 0) of
          NONE => NONE
        | SOME i => SOME (~i)
    else
      if size(s) = 0 then
        NONE
      else
        iFSAux(0, 0)
  end

intFromString("~12")
~> if size("~12") > 0 andalso
String.sub("~12",0) = #"~"
then ... else ...
~> if 2 > 0 andalso #"~" = #"~" then ...
else ...
~> if size("~12") = 1 then NONE else case...
~> if 3 = 1 then NONE else case...
~> case iFSAux(1, 0) of NONE => NONE| SOME i
=> SOME (~i)

........

~> case SOME 12 of NONE => NONE| SOME i =>
SOME (~i)
~> SOME (~12)
```

Hur fungerar `iFSAux`?

```
fun iFSAux(n, acc) =
  if n = size(s) then
    SOME acc
  else
    let
      val digit =
          ord(String.sub(s,n))-ord(#"0")
    in
      if digit < 0 orelse digit > 9 then
        NONE
      else
        iFSAux(n+1, acc*10+digit)
    end

(s = "~12")

iFSAux(1,0)
~> if 1 = size("~12") then SOME 0 else let..
~> let val digit = ord(String.sub("~12",1))-
ord(#"0") in ... end
~> let val digit = ord(#"1")-ord(#"0") in
... end
~> let val digit = 49-48 in ... end
~> let val digit = 1 in ... end
~> if 1 < 0 orelse 1 > 9 then NONE else
iFSAux(1+1, 0*10+1)
~> iFSAux(2,1)
~> if 2 = size("~12") then SOME 1 else let..
~> let val digit = ord(String.sub("~12",2))-
ord(#"0") in ... end
~> let val digit = ord(#"2")-ord(#"0") in
... end
~> let val digit = 50-48 in ... end
~> let val digit = 2 in ... end
~> if 2 < 0 orelse 2 > 9 then NONE else
iFSAux(2+1, 1*10+2)
~> iFSAux(3,12)
~> if 3=size("~12") then SOME 12 else let..
~> SOME 12
```

Mer om konvertering

ML innehåller en mängd biblioteksrutiner för konvertering, se kursboken appendix D.9.

Vårt exempel `intToString` är samma som biblioteksfunktionen `Int.toString`.

Vårt exempel `intFromString` liknar biblioteksfunktionen `Int.fromString`.

I verkligheten är konvertering från strängar besvärligt eftersom man i allmänhet vill konvertera en sträng med olika typer av information blandade efter varandra, t.ex: `"beräkna 1+23*4 slut"`. Ur denna sträng skulle man vilja få:

– strängen `"beräkna"`
– talet `1`
– tecknet `+`
– talet `23`
– tecknet `*`
– talet `4`
– strängen `"slut"`

Dessutom vill man kanske ha `1+23*4` representerat som ett träd.

Detta problem kallas *parsning*.

En kalkylator i ML

```
fun calc() =
  let
    fun calcRead() =
    (print("Skriv ett tal: ");
      let
        val t = inputLine(stdIn)
      in
        case intFromString(
           String.substring(t,0,size(t)-1))
         of
          NONE => (print("Felaktigt tal\n");
                   calcRead())
        | SOME i => i
      end)
    fun calcOp(x) =
          (print "Skriv en operation: ";
           case inputLine(stdIn) of
             "q\n" => ()
           | "c\n" => calcOp(calcRead())
           | "+\n" =>calcPrint(x+calcRead())
           | "-\n" =>calcPrint(x-calcRead())
           | "*\n" =>calcPrint(x*calcRead())
           | "/\n" =>
                calcPrint(x div calcRead())
           | _    =>
             (print("Felaktig operation\n");
              calcOp(x)))
    and calcPrint(x) =
          (print("Resultatet är: ");
           print(intToString(x)^"\n");
           calcOp(x))
  in
    calcOp(calcRead())
  end
```

Körning av kalkylatorn

```
- calc();
Skriv ett tal: 12
Skriv en operation: -
Skriv ett tal: 2
Resultatet är: 10
Skriv en operation: /
Skriv ett tal: 2
Resultatet är: 5
Skriv en operation: *
Skriv ett tal: 2
Resultatet är: 10
Skriv en operation: +
Skriv ett tal: 2
Resultatet är: 12
Skriv en operation: c
Skriv ett tal: 1
Skriv en operation: +
Skriv ett tal: 2
Resultatet är: 3
Skriv en operation: q
> val it = () : unit
```