Institutt for informatikk

Funksjonell Programmering i Standard ML

Bjørn Kristoffersen

Kompendium 61

Høstsemesteret 1995

Forord

Dette kompendiet er en introduksjon til funksjonell programmering i Standard ML beregnet for kurset IN 211: *Programmeringsspråk*. I IN211 inngår det en bolk om funksjonell programmering som undervises over 3 til 4 dobbelt-timer. Tidligere har undervisningen vært basert på Lisp og APE. Sistnevnte er et "tavlespråk" utviklet av Olaf Owe [TAA94].

Høsten 1994 tok vi i bruk Standard ML som et alternativ til APE; begge er sterkt typede språk. Motivasjonen for dette var å gi studentene en sjanse til å prøve ut typet funksjonell programmering på maskinene. Standard ML ble dermed et naturlig valg fordi det finnes flere gode implementasjoner, blant annet Standard ML of New Jersey som vi benyttet.

Grunnen til at dette kompendiet er skrevet skyldes for det første at læreboka [GJ87] i liten grad dekker typet funksjonell programmering generelt og Standard ML spesielt. For det andre består annet skriftlig materiale om Standard ML stort sett av hele lærebøker, eller også av rimelig komprimerte innføringer myntet på studenter med en litt annen bakgrunn enn den våre studenter har. Forøvrig finnes det en innføring i ML myntet på kurset IN 105, skrevet av Knut Omang [Oma94]. Vi går noe lenger enn denne.

Kompendiet er en generell innføring i Standard ML, og er ment å kunne bli benyttet også i andre kurs enn IN211. En foreløpig utgave av dette materialet fungerte høsten 1994 som støttelitteratur i hovedfagskurset *Utvalgte emner innen funksjonell programmering*.

Bjørn Kristoffersen
Oslo, 31. desember 1994 [1]

[1] Noen få endringer og rettelser er gjort høsten 95 av Olaf Owe.

Innhold

1 Introduksjon **2**

2 Uttrykk **2**

3 Navn og Deklarasjoner **6**

4 Funksjoner **8**

5 Imperative konstruksjoner **10**

6 Brukerdefinerte typer **12**

7 Polymorfi **14**

8 Høyere ordens funksjoner **15**

9 Modularisering **18**

A Et utsnitt av Standard ML **22**

1 Introduksjon

Standard ML (SML) er et *interpreterende* språk. Det betyr at interaksjon med SML foregår ved å taste inn uttrykk som så blir evaluert og skrevet ut på skjermen. For mindre oppgaver er det dermed ikke nødvendig å ha tilgang på en separat editor for å benytte SML. Den anbefalte måten å arbeide på er likevel å editere arbeidsuttrykk i Emacs, som deretter limes inn i SML-vinduet. Dersom filen med SML-kode gis extension `.sml`, går Emacs automatisk inn i SML-modus.

Det finnes flere ulike implementasjoner av SML. Vi skal benytte **Standard ML of New Jersey**. Heretter bruker vi bare forkortelsen SML-NJ. Sammen med implementasjonen av SML-NJ følger det endel nyttig dokumentasjon, blant annet en beskrivelse av innebygde typer og funksjoner og diverse verktøy (som for eksempel en variant av YACC for SML-NJ). Ta en titt på området:

```
/local/src/sml/93/doc/manual
```

Som lærebok i SML for de som "kan programmere" anbefales [Pau91]. Det finnes også en nybegynnerbok i programmering som benytter SML [Wik87].

SML-NJ startes fra Unix ved kommandoen `sml`. Systemet skriver en velkomstmelding avsluttet med bindestrek. Bindestreken forteller oss at SML-NJ er klar for å ta imot kommandoer :

```
menja 2>sml
Standard ML of New Jersey, Version 0.93, February 15, 1993
val it = () : unit
-
```

Nå er det bare å taste i vei (selv om du nok vil oppnå raskere fremskritt ved å lese videre først). For å avslutte systemet, taster du *CTRL-D*.

2 Uttrykk

SML-NJ som kalkulator

Vi kan bruke SML-NJ som kalkulator. Følgende uttrykk beregner antall rekker på en tippekupong:

```
- 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3 * 3;
val it = 531441 : int
```

Husk å avslutte uttrykk med semikolon. Kvitteringen fra SML-NJ forteller oss at resultatet av beregningen er `531441`, og at det er et heltall (det vil si av type `int`). Nøkkelordet `val`

(for value) betyr at `531441` er en *verdi*. Identifikatoren `it` bryr vi oss ikke om ennå. Hvem sa at funksjonelle språk ikke kan brukes til noe nyttig?

Negative heltall skrives ved å prefikse med tilde, for eksempel `~3`. Tilde er valgt som negasjonstegn for å skille det fra subtraksjon.

```
- 2 - ~3;
val it = 5 : int
```

Infix notasjon

Operatorer som addisjon `+` og multiplikasjon `*` kan skrives infix, det vil si at operatoren settes mellom sine to parametre som i `2*2`. Legg merke til at vi i fotball-eksemplet fikk lov til å kjede sammen flere operatoruttrykk uten bruk av parenteser. Noen ganger er innsetting av parenteser signifikant for resultatet.

```
- 2+3*4;
val it = 14 : int
- (2+3)*4;
val it = 20 : int
```

Det første utrykker svarer til å sette parenteser `2+(3*4)`, og viser at multiplikasjon binder sterkere enn addisjon. Tilsvarende kan assosiativitet av enkelte operatorer spille en rolle for betydningen av et uttrykk.

```
- (2 div 2) div 2;
val it = 0 : int
- 2 div (2 div 2);
val it = 2 : int
```

Hva er verdien til `2 div 2 div 2`, og hva forteller det deg om assosiativiteten til `div`? Ytterligere noen infix-operatorer over heltall er innebygget: `mod`, `quot`, `rem`, `max`, `min`. Dessuten har vi tilgjengelig vanlige relasjonsoperatorer (også infix): `>`, `>=`, `<` og `<=`. Finn ut hva de gjør, og forsøk å få en følelse med hvilke presedens- og assosiativitets-regler som gjelder. SML-NJ tillater programmereren å definere sine egne infix-operatorer. Det lar vi ligge.

Andre innebygde typer

Nå er det ikke slik at alt er bare heltall i SML.

```
- 3.14159 * 5.0 * 5.0;
val it = 78.53975 : real
```

Legg merke til at * (og også en del andre operatorer) fungerer både over heltall og reelle tall. Fordi heltall og reelle tall blir representert forskjellig i maskinen har SML-NJ to ulike implementasjoner av multiplikasjon. Hvilken som velges avhenger av typen til parameterne. Vi sier * er en *overlastet* operator. Foruten tall finnes det også sannhetsverdier og tekststrenger. Her er noen eksempler.

```
- not(true);
val it = false : bool
- if 1<2 then "b"^substring("Hallo",1,3) else "Hei";
val it = "ball" : string
- chr (ord ("a") + 1);
val it = "b" : string
```

Av og til går det galt, også i ML. Inspirert av tidligere suksess forsøker vi å finne antall forskjellige Lotto-rekker. Etter noen kombinatoriske resonnementer kommer vi fram til følgende formel:

```
- (34 * 33 * 32 * 31 * 30 * 29 * 28) div (2 * 3 * 4 * 5 * 6 * 7);
uncaught exception Overflow
```

Et nedslående resultat på flere måter. (Lurer du på resultatet kan du regne med reelle tall i stedet.) Identifikatoren `Overflow` i Lotto-eksemplet er en *exception constructor*, eller unntakskonstruktør. Det er mulig å definere egne unntak. Det er også mulig å fange opp unntak, noe som typisk gjøres ved kall av "farlige" funksjoner. Vi kommer tilbake til unntakshåndtering senere.

Records og tupler

Records skriver vi slik:

```
- {Name = "Bjørn Kristoffersen", Age = 30};
val it = {Age=30,Name="Bjørn Kristoffersen"}
       : {Age:int, Name:string}
```

`Name` og `Age` er *komponent-navn* (labels). Legg merke til at notasjonen for record-typer ligner på notasjonen for record-verdier. Den viktigste operasjonen på records er å plukke ut delkomponenter. Det gjøres med en litt uvant syntaks.

```
- #Name {Name = "Bjørn Kristoffersen", Age = 30};
val it = "Bjørn Kristoffersen" : string
```

For å plukke ut komponenten svarende til label `L` fra record `r`, skriver vi altså `#L r`. Fordi recorder er verdier kan vi ha recorder som delkomponenter i andre recorder.

```
- {Name = "James", Car = {Id = "007", Year = 1990}};
val it = {Car={Id="007",Year=1990},Name="James"}
  : {Car:{Id:string, Year:int}, Name:string}
```

To record-typer er like kun hvis både komponent-navn og typene til komponentene er like. Rekkefølgen på komponentene i en record spiller imidlertid ingen rolle (som du kanskje har observert ved å studere utskriften fra SML-NJ).

```
- {Name = "James", Id = 7} = {Id = 7,Name = "James"};
val it = true : bool
```

Tupler er en spesiell form for records der komponentene aksesseres ved hjelp av deres posisjon.

```
- ("James",59);
val it = ("James",59) : string * int
```

Dette er ekvivalent med følgende litt mer omstendelige skrivemåte:

```
- {1 = "James",2 = 59};
val it = ("James",59) : string * int
```

Aksessering av tuppelkomponenter kan gjøres som for records, for eksempel kan vi hente ut `"James"` ved `#1 ("James",59)`. Vi skal senere se på en annen måte å dekomponere tupler på som er mer hensiktsmessig.

Lister

Lister er innebygget i SML, og skrives for eksempel slik:

```
- [1,2,3];
val it = [1,2,3] : int list
```

Legg merke til at applikasjon av typekonstruktører (som for eksempel `list`) skrives postfiks, det vil si `int list` i stedet for `list(int)` (som kanskje ville være mer intuitivt). Vi kan ha lister der komponentene er av andre typer enn tall (forsøk med strenger, reelle tall, records, lister av records, osv). For enhver gitt liste må imidlertid alle komponentene i listen være av samme type.

```
- [1,"Feilmelding i vente"];
std_in:18.1-18.25
  Error: operator and operand don't agree
         (tycon mismatch)
```

```
  operator domain: int * int list
  operand:         int * string list
  in expression:
    1 :: "Feilmelding i vente" :: nil
```

Den siste linjen i feilmeldingen var kanskje noe overraskende (hvis du orket å lese den). Uttrykket vi tastet inn ble skrevet ut på en annen måte. Til det er å si at notasjonen med hakeparenteser egentlig er en avledet form. Lister representeres internt ved hjelp av `nil` og `::` ("cons"), der førstnevnte er et navn på en tom liste og sistnevnte bygger en større liste fra et element og en liste. For utskrift benytter SML-NJ stort sett den mer intuitive notasjonen.

```
- 1 :: 2 :: 3 :: nil;
val it = [1,2,3] : int list
```

Lister kan dekomponeres ved hjelp av funksjonene `hd` ("head") og `tl` ("tail"), og settes sammen med funksjonen `@` ("concat"):

```
- hd [1,2,3];
val it = 1 : int
- tl [1,2,3];
val it = [2,3] : int list
- [1,2,3] @ [4,5,6];
val it = [1,2,3,4,5,6] : int list
```

3 Navn og Deklarasjoner

Deklarasjoner utvider omgivelsen

For å slippe og skrive `3.14169` alle steder vi trenger å referere pi (med de muligheter for feil som det medfører), introduserer vi en *variabel*.

```
- val pi = 3.14159;
val pi = 3.14159 : real
```

Vi sier at `pi` blir *bundet* til verdien `3.14159`. Effekten av en slik *deklarasjon* er å utvide *omgivelsen* du til enhver tid jobber med. Dette ble noen nye begreper som nok trenger forklaring. Enkelt sagt, og ikke særlig presist, er en omgivelse en samling navnebindinger. Navn kan bindes til verdier, funksjoner (som i SML forøvrig er spesielle verdier), typer og flere andre entiteter. For eksempel er `*` med i den omgivelsen du har tilgjengelig ved oppstart av SML-NJ. Navnet `*` er da bundet til funksjonen multiplikasjon.

Navn som er bundet i omgivelsen kan benyttes i uttrykk. Det betyr at vi kan ta i bruk `pi`, for eksempel til å deklarere en ny variabel:

```
- val etAreal = pi * 5.0 * 5.0;
val etAreal = 78.53975 : real
```

Derimot er `nyVariabel` ikke kjent for SML-NJ.

```
- nyVariabel;
std_in:31.1-31.10 Error:
  unbound variable or constructor: nyVariabel
```

Redeklarasjoner

Det er mulig å *redeklarere* kjente navn. Navnet `abs` er ved oppstart bundet til absoluttverdi-funksjonen. Kanskje ønsker vi i stedet å la `abs` være en forkortelse for navnet til en ansatt ved instituttet (som interesserer seg for SML):

```
- val abs = "Anne B. Salvesen";
val abs = "Anne B. Salvesen" : string
```

Den nye deklarasjonen av navnet `abs` skygger for den gamle, på samme måte som lokale variable skygger over globale i Pascal eller Simula. La oss sjekke dette.

```
- abs (~3);
std_in:0.0-0.0 Error: operator is not a function
  operator: string
  in expression:
    abs (~3)
```

For SML-NJ er nå `abs` en tekststreng, og ikke en funksjon. Vi får dermed en *typefeil*. Du kan tenke på dialogen med systemet som en blokk-struktur, der hver nye deklarasjon starter et nytt skop. Når vi returnerer til operativsystemet avsluttes alle skopene.

Deklarasjonen av `etAreal` i paragrafen over benyttet `pi`. Hva om vi redeklarerer `pi`?

```
- val pi = 1.0;
val pi = 1.0 : real
- etAreal;
val it = 78.53975 : real
```

Vi ser at redeklarasjonen av `pi` ikke har noen betydning for verdien av `etAreal`. Grunnen er at sistnevnte er bundet til *verdien* av uttrykket `2.0 * pi * 5.0` (i omgivelsen vi hadde ved deklarasjon av `etAreal`), og ikke selve uttrykket.

Lokale deklarasjoner

Noen ganger ønsker vi å benytte et navn i en begrenset del av koden vår. Da benytter vi et let-uttrykk. Et eksempel hentet fra formelsamlingen følger. (Fordi vi her trenger flere linjer, kvitterer SML-NJ med et likhetstegn etter hver gang vi har tastet vognretur for å vise at systemet forventer mer kode.)

```
- val areal =
=  let
=    val a = 9.5
=    val b = 7.0
=    val c = 16.1
=    val s = (a + b + c) / 2.0
=  in
=    sqrt (s * (s - a) * (s - b) * (s - c))
=  end;
val areal = 14.3583564519063 : real
```

Variablene `a`, `b`, `c` og `s` er tilgjengelige i `let`-kroppen slik vi ønsker. Ved utgang av let-uttrykket er de ikke lenger tilgjengelige. Sjekk dette! Verdien av uttrykket er forøvrig arealet av en trekant med sider `9.5`, `7.0` og `16.1` i en passende måleenhet.

4 Funksjoner

Myk start

Dette skulle handle om funksjonell programmering, så det er vel på tide å definere noen funksjoner.

```
- fun divides (n:int,i:int):bool = n mod i = 0;
val divides = fn : int * int -> bool
- fun fac (n:int):int = if n=1 then 1 else n*fac(n-1);
val fac = fn : int -> int
- fun gcd (n:int,m:int) = if m=0 then n else gcd(m,n mod m);
val gcd = fn : int * int -> int
```

Legg merke til at SML-NJ selv fant resultat-typen til `gcd`. Senere skal vi se at vi også kan utelate typingen av de formelle variablene. Siden disse funksjonene er rekursive har vi allerede sett hvordan funksjoner appliseres. For ordens skyld:

```
- fac (10);
val it = 3628800 : int
```

Non-lokale variable

La oss definere en funksjon som benytter en non-lokal variabel.

```
- val pi = 3.14159;
val pi = 3.14159 : real
- fun areal (r:real) = pi * r * r;
val areal = fn : real -> real
- areal (3.0);
val it = 28.27431 : real
```

Inne i kroppen til `areal` har vi altså tilgjengelig den non-lokale `pi`. Som ventet. Hva om vi redeklarerer navnet `pi`?

```
- val pi = 1.0;
val pi = 1.0 : real
- areal (3.0);
val it = 28.27431 : real
```

Det samme resultatet. Det betyr at `areal` benytter verdien til den variabelen `pi` som var synlig da funksjonen ble definert. At vi redeklarerer `pi` betyr ikke at vi "tilordner" `pi` en ny verdi (som i imperative språk), men at vi deklarerer en *ny* variabel som tilfeldigvis heter det samme som en variabel allerede i omgivelsen.

Funksjoner som parametre

Anta vi skal beregne summen av tallene i en liste. Oppgaven går ut på å "dytte pluss'er" inn mellom elementene i en liste. For listen `[2,7,3]` ønsker vi å beregne `2+7+3`. Er vi litt forutseende kan vi tenke oss å gjøre det samme med for eksempel multiplikasjon, eller en vilkårlig binær operator over tall. Funksjonen `fold` gjør dette, der parameteren `d` (for default) tillater oss å variere "startverdi":

```
- fun fold (f:int*int->int,d:int,l:int list):int =
=     if l = [] then d
=     else if tl(l) = [] then f(d,hd(l))
=     else f(hd(l),fold(f,d,tl(l)));
```

Produktet av oddetallene fra `1` til `9` beregnes nå slik:

```
fold(op*,1,[1,3,5,7,9]);
val it = 945 : int
```

Notasjonen `op*` benyttes fordi `*` er definert som en infix-operator mens den formelle parameteren `f` i funksjonen `fold` benyttes på vanlig prefiks måte. Navnet `op*` er bundet til

den samme funksjonen som `*`, men kan benyttes prefix, for eksempel som `op*(2,2)`. Forsøk gjerne dette med noen andre infix-operatorer også. I neste kapittel skal vi se hvordan funksjonen `fold` kan formuleres mer elegant ved hjelp av case-uttrykk.

Oppgave. Lag funksjoner:

```
- fun filter (f:int->bool,l:int list):int list = ...
- fun map (f:int->int,l:int list):int list = ...
```

Verdien til `filter(f,l)` skal være listen av elementer `x` fra `l` der `f(x)` er `true`. Tilsvarende skal verdien til `map(f,l)` være listen vi får ved å anvende `f` på hvert element i `l`.

Funksjoner som `map`, `filter` og `fold` kan gjøres langt mer generelle ved å løsne på (de unødvendig strenge) typekravene. Vi kommer tilbake til dette.

5 Imperative konstruksjoner

Dette avsnittet benytter konstruksjoner som ikke er beskrevet av grammatikken i tillegg A.

SML har tilordning, løkker, pekere og arrays. Vil man skrive imperativ kode går det altså aldeles utmerket, og det er mulig å kombinere funksjonell og imperativ programmering fritt. Her er en funksjon som beregner kvadratroten (med rest) av heltall. Koden er omarbeidet til SML fra [Dah92].

```
- fun sqrt (n : int) =
=   let
=     val x = ref 0
=     val y = ref n
=     val q = ref 1
=   in
=     while !q <= n do q := !q * 4;
=     while !q <> 1 do
=      (q := !q div 4;
=       if !y < !x + !q then
=         x := !x div 2
=       else
=         (y := !y - (!x + !q); x := !x div 2 + !q));
=     (!x,!y)
=   end;
- val sqrt = fn : int -> int * int
```

Kaller vi sqrt(n) får vi ut et par (x,y) der $x^2 \leq n \leq (x+1)^2$ og $y = n - x^2$. For eksempel:

```
- sqrt (10);
val it = (3,1) : int * int
```

Selv om koden over kan minne om språk i Algol-familien er det på sin plass med et par kommentarer. For det første skiller SML-NJ *ikke* syntaktisk mellom uttrykk og setninger. Det betyr for eksempel at både tilordninger og while-løkker er *uttrykk*. Ut av slike "imperative" uttrykk kommer kun det tomme tuppelet `()`. Effekten er altså å oppdatere tilstanden. For det andre benyttes parenteser for å omslutte blokker i stedet for begin/end-par. For det tredje må variable man vil "oppdatere" være *referanser*. Referanser opprettes ved hjelp av operatoren `ref`. Se på:

```
- val x = ref 1;
val x = ref 1 : int ref
- x := !x + 1;
val it = () : unit
- x;
val it = ref 2 : int ref
```

Først deklarerer vi `x` til å være en *referanse* (eller peker) til verdien `1`. Legg merke til at `ref` *navngir* både en funksjon og en type. Typen til `x` er `int ref`. I det andre uttrykket kan det se ut som om vi øker "verdien" av `x` med `1`. Det er bare nesten riktig. På venstresiden av `:=` skriver vi bare `x`, mens vi på høyresiden skriver `!x`. Førstnevnte gir oss verdien av `x`, som jo er en referanse til verdien `1`. Sistnevnte gir oss selve verdien `1`. Effekten av tilordningen er altså å øke verdien i den lokasjonen som verdien av `x` peker på. I tradisjonelle imperative språk er dette skjult bak kulissene ved at vi tolker forekomster av en variabel på venstresiden av tilordninger som adressen til variabelen, og forekomster på høyresiden som verdien pekt på av tilhørende adresse.

Referanser kan være delkomponenter i større datastrukturer som for eksempel records.

```
- val agent = {Name = "James", Age = ref 59};
val agent = {Age=ref 59,Name="James"}
          : {Age:int ref, Name:string}
(#Age agent) := !(#Age agent) + 1;
val it = () : unit
- agent;
val it = {Age=ref 60,Name="James"}
       : {Age:int ref, Name:string}
```

Også 007 blir altså eldre.

6 Brukerdefinerte typer

Oppramstyper

De fleste kjenner oppramstypen fra Pascal. Boolske verdier kan modelleres ved en oppramstype.

```
- datatype bool = true | false;
datatype  bool
  con false : bool
  con true : bool
```

Kvitteringen fra SML-NJ uttrykker at verdimengden til datatypen `bool` er *spent ut* av to *konstruktører* `true` og `false`. Du kan lese `|` som "eller". Analogien til bruk av `|` i BNF-grammatikker er nyttig. Typen `bool` er forøvrig predefinert i SML.

Vi kan definere funksjoner over oppramstyper ved hjelp av *case-uttrykk*. Negasjon kan defineres slik:

```
- fun not (b:bool) = case b of true => false | false => true;
val not = fn : bool -> bool
```

Det som står til venstre for `=>` i *case-alternativene* kalles for *mønstre* (eng: patterns). De er bygget opp fra konstruktører og variable. Uttrykket vi gjør *case-oppspalting* på (her `b`) trenger ikke å være en enkelt variabel:

```
- fun xor (b1,b2) =
      case (b1,b2) of
        (true,false) => true
      | (false,true) => true
      | _            => false;
val xor = fn : bool * bool -> bool
```

Legg merke til at tuppel-parentesene er konstruktører. Det spesielle symbolet _ kan brukes for å fange opp vilkårlige verdier (wildcard). En alternativ skrivemåte for siste alternativ i case-uttrykket er: `(_,_)=>false`.

Definer typen av ukedager, og definer en boolsk funksjon som tar en ukedag som parameter og avgjør om det er en arbeidsdag.

Rekursive typer

Oppramstyper kan generaliseres ved å tillate konstruktørene å ta argumenter. Vi kan lage *rekursive* typer. La oss representere binærtrær som en rekursiv type. For eksempel kan vi la løvnodene inneholde tall.

```
- datatype bintree = Tip of int | Tree of bintree * bintree;
datatype  bintree
  con Tip : int -> bintree
  con Tree : bintree * bintree -> bintree
```

La oss definere en funksjon som projiserer et tre til en liste:

```
- fun proj t = case t of
=     Tip(n) => [n]
=   | Tree(l,r) => proj(l) @ proj(r);
val proj = fn : bintree -> int list
- proj(Tree(Tree(Tip(1),Tip(2)),Tip(3)));
[1,2,3] : int list
```

Legg merke til hvor enkelt vi kan lage verdier av rekursive typer. I et tradisjonelt imperativt språk må vi typisk benytte pekere for å oppnå det samme.

Hva med *oppdatering* av store datastrukturer? Kan vi leve med SML i slike situasjoner? La oss definere en funksjon som setter inn et element lengst til venstre i et tre av type `bintree`.

```
- fun ins(n,t) = case t of
=     Tip(n') => Tree(Tip(n),Tip(n'))
=   | Tree(l,r) => Tree(ins(d,l),r);
val ins = fn : int * bintree -> bintree
```

Noen vil innvende at slik programmering er håpløst ineffektivt. For å sette inn en ny verdi i et tre, lager vi et helt nytt tre! Dette er imidlertid ikke helt sant. `bintree` kan implementeres som en pekerstruktur (av kompilatoren). Dermed holder det å generere på nytt de cellene vi påtreffer fra roten i treet ned til verdien vi skal sette inn. Resten av treet er uendret og bør ikke genereres ikke på nytt [Dah92]. Hvis det ikke er effektivt nok har man tilgang til både pekere og imperative konstruksjoner i SML.

Partielle funksjoner

Av og til gir en funksjon kun mening når den appliseres med visse verdier i en type. For eksempel får vi en feil dersom vi forsøker å ta "hodet" til den tomme lista:

```
- hd([]);
uncaught exception Hd
```

Vi sier `hd` er er *partiell funksjon*; den er kun definert over ikke-tomme lister. En fordel med funksjonsdefinisjon ved case-oppspalting er at systemet kan kontrollere at vi får med alle muligheter, og i motsatt fall gi en advarsel. La oss definere en funksjon som returnerer venstre grein i binærtrær som definert over:

```
- fun left(t:bintree) = case t of Tree(l,_)=>l;
std_in:0.0-0.0 Warning: match nonexhaustive
          Tree (l,_) => ...
val left = fn : bintree -> bintree
```

SML-NJ gjør oss oppmerksomme på at vi mangler en grein i case-uttrykket, men aksepterer dog deklarasjonen. Følgende viser at `left` er en partiell funksjon:

```
- left(Tip(1));
uncaught Match exception std_in:0.0-0.0
```

`Match` er et systemdefinert unntak som opptrer når SML-NJ evaluerer case-uttrykk, og ikke finner en venstreside som passer. SML-NJ kontrollerer forøvrig også at venstresidene i case-uttrykk ikke er *overlappende*.

Både `Hd` og `Match` er unntakskonstruktører. Vi kan definere nye unntak, "løfte" (eng:raise) unntak og fange unntak. Her er et eksempel som illustrerer alle tre varianter:

```
- exception Left; (* definerer *)
exception Left
- fun left(t:bintree) = case t of
      Tip(_) => raise Left (* løfter *)
    | Tree(l,_) => l;
val left = fn : bintree -> bintree
- left(Tip(1)) handle Left=>Tip(0); (* fanger *)
val it = Tip 0 : bintree
```

7 Polymorfi

I eksemplet med binærtrær over er det lett å se at hverken `proj` eller `ins` baserer seg på kunnskap om representasjonen av element-typen `int`. De vil fungere for vilkårlige typer. Kun strukturen på treet er av betydning.

Pascal (for å ta et eksempel) er sterkt typet. Og uanstendig stivt. Det er ikke mulig å lage en generell liste. Vi må programmere lister av tall, lister av tekst, lister av persondata. For ikke å snakke om lister av lister av tall. Likeledes må vi for hver liste-type lage en ny lengde-funksjon. Problemet er ikke at Pascal er sterkt typet, men at det er *monomorft*. Ingen prosedyrer fungerer på mer enn en type.

SML er et *polymorft* språk. Polymorfi betyr "å anta mange former". En liste kan opptre i mange former ved at elementene kan ha ulike typer. Men en liste er likevel en liste. La oss gjenta eksemplet med trær, men denne gangen polymorft. I stedet for `int` benytter vi en *typevariabel* `'a`. (Typevariable prefikses med en apostrof for å skille dem fra konkrete typer.)

```
- datatype 'a bintree =
      Tip of 'a
    | Tree of 'a bintree * 'a bintree;
datatype  'a bintree
  con Tip : 'a -> 'a bintree
  con Tree : 'a bintree * 'a bintree -> 'a bintree
- fun proj (t:'a bintree) =
      case t of
        Tip(d) => [d]
      | Tree(l,r) => proj(l) @ proj(r);
val proj = fn : 'a bintree -> 'a list
```

Legg merke til at koden for `proj` er uendret, men typen er mer generell. Det samme vil gjelde for `ins`. Et par eksempler på bruk av den polymorfe `proj`:

```
- proj(Tree(Tree(Tip(1),Tip(2)),Tip(3)));
[1,2,3] : int list
- proj(Tree(Tip(true),Tree(Tip(false),Tip(true))));
[true,false,true] : bool list
```

Polymorfi gir *gjenbruk* av kode. Vi slipper å skrive samme ting flere ganger. Med mindre kode blir det også lettere både å forstå og endre program, ganske enkelt fordi vi har mindre tekst å forholde oss til. Fordi vi kan klare oss med én "versjon" av funksjoner som `inf`, avtar dessuten *navneuniverset* i størrelse. Mang en Pascal-programmeren har opplevd "navnekriser", og tydd til løsninger som `LengdeAvPersonListe`. Polymorfi gjør sterk typing fleksibel.

8 Høyere ordens funksjoner

Høyere ordens funksjoner er basert på at funksjoner er data på samme måte som tall og tekst er det. Det betyr at de kan sendes som parametre, mottas som returverdier og bindes til variable. Med et slagord sier vi at "funksjoner er fullverdige borgere".

Operasjoner på lister

Høyere ordens funksjoner er særlig nyttige i forbindelse med brukerdefinerte typer (og polymorfi). Vi har allerede sett på en enkel variant av `map`. Vi lovet å gjøre `map` mer nyttig ved å løsne på typekravene. Her er den generelle definisjonen:

```
- fun map f l = case l of
=     [] => []
=   | x::xs => (f x)::(map f xs);
val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

Funksjonen `map` tar en funksjon `f` fra `'a` til `'b` for *vilkårlige* typer `'a` og `'b`, og en liste `l` av type `'a list`, og returnerer listen av `f` anvendt på hvert element i `l`. Det betyr at resultatet er en verdi av type `'b list`. Legg merke til at systemet selv finner ut typen; de formelle variablene til `map` er ikke typedeklarert. Dette kalles *typeinferens*.

Vi kan nå benytte `map` på ulike måter. For eksempel kan vi gitt en liste av lister plukke ut det første elementet fra hver av listene.

```
- map hd [[1,2,3],[50],[333,444,555]];
val it = [1,50,333] : int list
```

Av og til trenger vi en funksjon kun én gang. I så fall trenger vi ikke å sette navn på den. SML tilbyr en spesiell notasjon for definisjon av slike "anonyme" funksjoner ved nøkkelordet `fn`. De benyttes ofte i forbindelse med høyere ordens funksjoner som `map`.

```
- map (fn x => x*2) [1,2,3];
val it = [2,4,6] : int list
```

Legg merke til at anonyme funksjoner ikke kan være rekursive (vi har ikke noe navn på dem).

Currying

Notasjonen for funksjonsapplikasjon er annerledes i SML enn i imperative språk: `f x` i stedet for `f(x)`. Notasjonen `f(x,y)` betyr altså *ikke* at `f` tar to parametre, men at den ene parameteren `(x,y)` er et par. Dette har vi til nå skjøvet under teppet ved å legge på (overflødige) parenteser.

Funksjonen `^:string*string->string` konkatenerer to strenger og er predefinert i SML. Vi kan definere en beslektet funksjon `concat` som følger:

```
- fun concat s1 s2 = s1 ^ s2;
val concat = fn : string -> string -> string
```

Vi sier `concat` er den *currierte* versjonen av `^`, etter logikeren H. F. Curry. (Kanskje burde vi heller beæret en herre ved navn Schönfinkel, men "schönfinkeliering" har vel aldri helt slått an.) La oss gjøre dette klart: funksjonen `concat` er en funksjon som tar en streng `s1` og returnerer en funksjon (som vi ikke har noe navn på) som tar en streng `s2` og returnerer strengen `s1 ^ s2`.

Med den nye versjonen av konkatenering dukker det dermed opp en ny mulighet. Vi kan benytte *resultatet* av å applisere `concat` med bare ett argument, som jo er en funksjon, som argument til andre funksjoner. For eksempel er resultatet av `concat "hei "` en funksjon som tar en streng `s` som argument og returnerer `"hei " ^ s`. Vi kan for eksempel tenke oss å konkatenere strengen "hei " med hvert enkelt av elementene i en liste:

```
- map (concat "hei ") ["Ole","Dole"];
val it = ["hei Ole","hei Dole"] : string list
```

Videre kan vi lage "`map`"-funksjoner over andre datastrukturer vi måtte finne på å definere, for eksempel `bintree`. SML tillater ikke overlasting av *brukerdefinerte* funksjoner, så vi kan dog ikke bruke navnet "`map`":

```
- fun BTmap f Tip(d) = Tip (f d)
    | BTmap f Tree(l,r) = Tree(BTmap f l,BTmap f r);
val BTmap = fn : ('a -> 'b) -> 'a bintree -> 'b bintree
```

Dronning-oppgaven

Dronning-oppgaven går ut på å plassere 8 dronninger på et sjakk-brett slik at ingen kan slå hverandre. Det betyr at det ikke må finnes 2 dronninger som er på samme rad, kolonne eller diagonal. La oss forsøke å skrive en funksjon som finner alle slike løsninger.

Etter noe tankevirksomhet innser vi at et dronning-utlegg kan representeres ved en liste av lengde 8, der element `i` angir hvilken rad dronningen i kolonne `i` befinner seg på. Dersom vi sørger for at enhver liste inneholder (alle) tallene fra 1 til 8 er det lett å se at vi i utgangspunktet bare betrakter posisjoner der alle rader og alle kolonner har nøyaktig én dronning.

Vi skal velge den opplagte algoritmen, som er først å generere alle permutasjoner av listen som inneholder tallene fra 1 til 8, og dernest fjerne de som har mer enn én dronning på noen diagonal. Fordi vi benytter lister kan vi enkelt generalisere problemet til `n` dronninger. Vi definerer en funksjon `fromTo` som (blant annet) kan lage listen `[1,2,...,n]`. Funksjonen `perm` lager alle mulige permutasjoner av en liste. Den bruker hjelpefunksjonen `comb` for å kombinere et element og en liste på alle mulige måter, og returnerer altså en liste av lister. I den forbindelse trenger vi også en funksjon `flat` som tar en lister av lister og lager én lang liste ved konkatenering.

Da gjenstår det å test på diagonaler. Anta vi har 2 dronninger i kolonnene `i` og `j`, der henholdsvis `rad(i)` og `rad(j)` angir raden de befinner seg i. Da vet vi at de befinner seg på samme diagonal hvis og bare hvis `|rad(i)-rad(j)| = |i-j|`. Vi skriver en funksjon `nodiag` som sjekker om en dronning lar seg plassere i venstre kant av en gitt liste ved å vandre rekursivt gjennom listen der vi akkumulerer avstanden (fra venstre kant) i en parameter. Det er nå lett å definere en funksjon `legal` som sjekker om (listerepresentasjonen av) et dronning-utlegg er lovlig.

Hovedfunksjonen kaller vi for `queen`. Den benytter `filter` med predikat `legal` for å fjerne ulovlige posisjoner fra listen av alle permutasjoner. Følgende samling av funksjoner implementerer dronning-oppgaven:

```
- fun curry f x y = f(x,y);
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
- fun fromTo(i,j) = if i<=j then i::(fromTo(i+1,j)) else [];
```

```
val fromTo = fn : int * int -> int list
- fun flat [] = []
    | flat (x::xs) = x @ (flat xs);
  val flat = fn : 'a list list -> 'a list
- fun comb x [] = [[x]]
    | comb x (y::ys) =
        (x::(y::ys)) :: (map (curry (op::) y) (comb x ys));
val comb = fn : 'a -> 'a list -> 'a list list
- fun perm [] = [[]]
    | perm (x::xs) =
        flat (map (comb x) (perm xs));
val perm = fn : 'a list -> 'a list list
- fun nodiag n x [] = true
    | nodiag n x (y::ys) =
        (abs(x-y) <> n) andalso (nodiag (n+1) x ys);
val nodiag = fn : int -> int -> int list -> bool
- fun legal [] = true
    | legal (x::xs) = (nodiag 1 x xs) andalso (legal xs);
val legal = fn : int list -> bool
- fun queen n = filter legal (perm (fromTo(1,n)));
val queen = fn : int -> int list list
```

La oss teste ut programmet. Vi forsøker først med et brett av størrelse 4.

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

Funksjonen `queen` finner *alle* posisjoner. Noen av disse vil imidlertid være *speilinger* av hverandre. Leseren oppfordres til å gjøre de nødvendige modifikasjoner for avskjæring.

Den valgte løsningen er svært lite effektiv. En bedre løsning ville være å sjekke på lovlige posisjoner *under* generering av permutasjonene. Se [Pau91] for en slik variant. Vi håper i det minste det var rimelig lett å *forstå* programmet.

Legg også merke til at flere av funksjonene vi benyttet er generelle operasjoner som vil finnes i et standard-bibliotek. Egentlig er det bare `nodiag`, `legal` og `queen` som er spesifikke for dronning-oppgaven.

9 Modularisering

Vi har til nå sett på hvordan ML egner seg for små program. Hva med strukturering av store system, eller mekanismer for "programming in the large"?

Signaturer

Anta vi skal definere begrepet *sekvensiell fil* i ML. Eksemplet er hentet fra [Dah92] (noe modifisert). En (sekvensiell) fil er en sekvens av *poster* med et *lesehode* som peker ut inneværende post. For å slippe å låse oss til en bestemt datastruktur og bestemte algoritmer ønsker vi å definere et *abstrakt grensesnitt* for filbegrepet. Det kan vi gjøre med en *signatur*. En signatur angir blant annet navnet og typen til operasjonene.

```
signature FIL =
  sig
    type fil
    type post
    exception Eof                (* End of File *)
    val emp  : fil               (* tom fil *)
    val ins  : fil * post -> fil (* sett inn post *)
    val del  : fil -> fil        (* fjern post *)
    val rew  : fil -> fil        (* spol tilbake *)
    val cur  : fil -> post       (* les post *)
    val more : fil -> bool       (* flere poster? *)
  end;
```

Vi har ikke definert typene `fil` og `post`. De er *opake* eller ugjennomsiktige typer. Det eneste vi krever er at funksjonene listet i grensesnittet blir definert. Vi stiller ingen krav til typen av poster.

Strukturer

I en prototype kan vi tenke oss å implementere filbegrepet uten å befatte oss med eksterne lagringsmedier. En enkel strategi er å betrakte `emp`, `ins`, `del` og `rew` som konstruktører. Det betyr i så fall at `cur` og `more` må defineres induktivt med hensyn på "rekkefølgen" disse operasjonene er utført i for en gitt fil. Vi velger å la postene i filen være tall (av type int).

En samling av definisjoner kan grupperes og navngis ved hjelp av en *struktur*. Vi kan oppnå (en viss grad av) skjuling dersom vi *beskranker* strukturen med en signatur, i vårt tilfelle med `FIL`. Dette er analogt med hvordan vi beskranker verdier med typer i "mikroprogrammering".

```
- structure Fil1 : FIL  =
    struct
      type post = int
      datatype fil =
        emp
      | ins of fil * post
      | del of fil
      | rew of fil;
```

```
      exception Eof;
      fun leftp emp = []
        | leftp (ins(F,x)) = (leftp F) @ [x]
        | leftp (del F) = leftp F
        | leftp (rew F) = [];
      fun rightp emp = []
        | rightp (ins(F,x)) = rightp F
        | rightp (del F) = tl (rightp F)
        | rightp (rew F) = leftp F @ rightp F;
      fun cur F = hd (rightp F);
      fun more F = case rightp F of [] => false | _ => true;
    end;
structure Fil1 : FIL
```

Definisjonen av typen `post` benytter nøkkelordet `type` i stedet for `datatype`. Det skyldes at `post` kun er et annet *navn* på typen `int`. Det viste seg nyttig å innføre to hjelpefunksjoner `leftp` og `rightp` som plukker ut henholdsvis venstre og høyre del av filen relativt til skrivehodet. Hjelpefunksjonene `leftp` og `rightp` er ikke synlige utad.

```
- Fil1.leftp;
std_in:3.1-3.10 Error: unbound variable or constructor:
                       leftp in path Fil1.leftp
```

Skrivemåten `Fil1.leftp` er et eksempel på et *sammensatt navn* som kan leses "(strukturen) `Fil1` sin (funksjon) `leftp`". Konstruktører og definerte funksjoner har samme status med hensyn på skjuling. Det som er synlig følger fra signatur-beskrankningen. Hadde vi lagt til en ekstra konstruktør, for eksempel `open_file`, ville den vært skjult på samme måte som de definerte funksjonene `leftp` og `rightp`.

Vi har imidlertid ikke oppnådd full abstraksjon av filbegrepet. Den systemgenererte *likhetsfunksjonen* `=` kan anvendes på verdier av type `fil`. Den bruker likhet på representasjon, og det er ikke hva vi ønsker. For eksempel vil vi at funksjonen `rew` skal være *idempotent*: `rew(rew F)=(rew F)`.

```
- Fil1.rew (Fil1.rew Fil1.emp) = Fil1.rew Fil1.emp;
val it = false : bool
```

For å bøte på dette fraviker SML-NJ den offisielle definisjonen av SML, og tilbyr en spesiell konstruksjon introdusert ved nøkkelordet `abstraction`, som kan settes inn der man kan bruke `structure`. Abstraksjon gir mulighet for fullstendig innkapsling. I vårt eksempel blir eneste mulighet for å skille to abstrakte verdier å benytte *observatørene* `cur` og `more`. Du bør forsøke å skille verdiene i uttrykket over fra hverandre med denne metoden.

Vi skylder her å gjøre oppmerksom på at SML har en egen mekanisme `abstype` (analogt med `datatype`) for å oppnå full sikkerhet. Denne delen av språket er imidlertid gammel, og det er ting som tyder på at framtidige utgaver av SML vil håndtere abstrakte datatyper i modulspråket, for eksempel ved hjelp av **abstraction**.

Det er en nyttig oppgave å implementere filbegrepet mer effektivt, for eksempel ved å benytte et par av lister `(l,r)`, der vi tenker oss at lesehodet "peker på" første element i høyre del `r`. Et mer ambisiøst prosjekt er å benytte de faktiske operasjonene for filbehandling i SML. Man bør da utvide grensesnittet med mulighet for navngiving av filer, åpning og lukking av filer osv.

Funktorer

Følgende er et naturlig grensesnitt for mengdebegrepet:

```
signature SET =
  sig
    type elm;
    type set;
    val  empty : set
    val  add   : set * elm -> set
    val  has   : set * elm -> bool
    val  subset: set * set -> bool
    val  eq    : set * set -> bool
  end;
```

Vi kan imidlertid ikke uten videre implementere disse funksjonene; vi trenger (som et minimum) *likhet* over elementene. La oss definere grensesnittet for en type med likhet.

```
signature EQ =
  sig
    type elm
    val  eq : elm * elm -> bool
  end;
```

Gitt en vilkårlig type som tilfredsstiller grensesnittet definert i signaturen `EQ`, kan vi nå definere mengdebegrepet. Det gjør vi ved hjelp av en *funktor*, som kan betraktes som en funksjon fra strukturer til strukturer.

```
functor mkSet (structure Eq:EQ):SET =
  struct
    type elm = Eq.elm
    type set = elm list
    val empty = []
    fun add (l,x) = x::l
    fun has (l,x) =
        case l of
          [] => false
        | (y::ys) => Eq.eq (x,y) andalso has (ys,x)
    fun subset (S1,S2) =
```

```
        case S1 of
          [] => true
        | x::xs => has (S2,x) andalso subset (xs,S2)
    fun eq(S1,S2) =
        subset (S1,S2) andalso subset (S2,S1)
end;
```

For eksempel vil heltallene kunne oppfylle signaturen `EQ`.

```
structure IntEq:EQ =
  struct
    type elm = int
    fun eq (x:int,y:int) = (x = y)
  end;
```

For å lage mengder av heltall, kan vi nå anvende `mkSet` med `IntEq`.

```
structure IntSet = mkSet (structure Eq=IntEq);
```

Funksjoner definert inne i strukturer kan aksesseres ved hjelp av prikk-notasjon, som for eksempel `IntEq.eq`.

```
val s1  = IntSet.add (IntSet.add (IntSet.empty,1),2);
val s2 = IntSet.add (IntSet.add (IntSet.empty,2),1);
IntSet.eq(s1,s2);
```

Den siste linjen i eksemplet er et uttrykk for å sjekke om de to mengdene `s1` og `s2` er like. Kvitteringen fra SML-NJ er `true`.

Forsøk å definere mengder av mengder av heltall. Må grensesnittet for mengder utvides? Anta dernest at element-typen har definert en passende ordningsrelasjon på seg. Kan du effektivisere implementasjonen, og hva må du i så fall gjøre med parameteren til `mkSet`?

A Et utsnitt av Standard ML

Dette tillegget definerer grammatikken til et utsnitt av SML. Først beskrives det som kan kalles leksikalske enheter. Det vil si hvilke nøkkelord som finnes, hva som er formatet for konstanter og identifikatorer. Deretter defineres grammatikken for uttrykk, deklarasjoner og typer.

I sammenhenger som dette kan det lett bli en konflikt mellom presisjon og fullstendighet på den ene siden og ønske om en pedagogisk framstilling på den andre. For det første var det nødvendig å definere et utsnitt av SML med tanke på den begrensede tiden som er tilgjengelig; IN 211 er ikke et SML-kurs. Det betyr både at enkelte konstruksjoner i språket

helt er tatt bort, og at andre er gitt i en forenklet utgave. Dernest er enkelte "detaljer" i den offisielle definisjonen [MTH90], slik som presedens mellom ulike alternativer og assosiativitet av operatorer hoppet bukk over. Endelig er grammatikken flatet ut ved hjelp av en utvidet utgave av BNF for å oppnå en mer kompakt beskrivelse. Kort fortalt benyttes metaparenteser ⟦og ⟧, valgfrie konstruksjoner angitt med et hevet spørsmålstegn $^{?}$, gjentakelse null eller flere ganger ved superskript stjerne *, og gjentakelse en eller flere ganger ved superskript pluss $^{+}$. De to sistnevnte kan i tillegg ha et subskript som forteller hva som skal adskille elementene som repeteres dersom det er to eller flere av dem. For eksempel angir $Ident^{*}_{,}$ en liste av null eller flere *Ident*'er adskilt av komma. Merk at metasymboler er skrevet i uthevet skrift. Se *Lokal Guide til LaTeX* for en presis beskrivelse av denne notasjonen.

I sum betyr dette at intuisjon er vektlagt på bekostning av presisjon sammenlignet med den offisielle definisjonen. Så er det bare å håpe at eventuelle feil (i betydningen at grammatikken her godtar programmer som ikke godtas av den offisielle grammatikken) er få og av liten betydning.

Leksikalske Enheter

De leksikalske enhetene som blir beskrevet her er reserverte ord, konstanter og identifikatorer. Dessuten gis formatet for kommentarer.

Reserverte Ord

Følgende er reserverte ord. Bortsett fra = kan de ikke benyttes som identifikatorer. Reserverte ord er i dette tillegget satt i **fete typer**.

abstype	**and**	**andalso**	**as**	**case**	**do**	**datatype**	**else**
end	**exception**	**fn**	**fun**	**handle**	**if**	**in**	**infix**
infixr	**let**	**local**	**nonfix**	**of**	**op**	**open**	**orelse**
raise	**rec**	**then**	**type**	**val**	**with**	**withtype**	**while**
eqtype	**functor**	**include**	**sharing**	**sig**	**signature**	**struct**	**structure**
( )	[ ]	{ }	, :	; ...	_ |	= =>	-> #

Konstanter

En heltallskonstant består av en ikke-tom sekvens av siffer, eventuelt prefikset med et negasjonstegn "~" . Reelle tall er heltall utvidet med et desimalpunktum "." fulgt av et heltall, og/eller utvidet med eksponentsymbolet "E" fulgt av et heltall. Enten desimaldelen eller eksponentdelen må være gitt for å skille reelle tall fra heltall. En strengkonstant er en sekvens av skrivbare tegn, blanke og escape-sekvenser innesluttet i hermetegn.

Her er noen eksempler:

Heltall 1 7813 ~2 0

Reelle tall 1.0 0.1 3.37E5 5E~7

Strenger “hello” “En ny linje\n” “Tab \t”

Kommentarer

Kommentarer angis med `(*` og `*)`, og kan være vilkårlig nestede. `(* Kommentar. *)`

Identifikatorer

En *kort identifikator* er enten en sekvens av bokstaver, tall, apostrofer “ ’ ” og underscores “ _ “ der første tegn må være en bokstav eller apostrof. Det er forskjell på små og store bokstaver. Eksempler på korte identifikatorer er `x`, `etMegetLangtNavn`, `to_navn`, `teller1`, `'a`. Det finnes også *symbolske* identifikatorer som kan inneholde ymse spesialtegn. Er du interessert kan du lese i definisjonen [MTH90] om dette. De korte identifikatorene er inndelt i kategorier ettersom hva slags entiteter de navngir, for eksempel verdier og typer. Syntaktisk er det kun typevariable som adskiller seg fra de andre ved at de må starte med en apostrof.[2] Formålet med å dele inn identifikatorene i klasser er at navneuniversene kan overlappe. Det betyr at vi kan ha en funksjon og en type som begge heter det samme. Grammatikken i definisjonen benytter egne metasymboler for de ulike kategoriene av identifikatorer og kan dermed uttrykke restriksjoner som blir ignorert her.

En (lang) *identifikator* er en sekvens av korte identifikatorer adskilt med punktum, for eksempel `BinaryTree.insertNode`. Lange identifikatorer blir benyttet for å angi entiteter (som funksjoner og typer) definert i moduler, for eksempel i standard-biblioteket.

Uttrykk, Deklarasjoner og Typer

BNF

$$
\begin{array}{lcl}
\mathit{Exp} & ::= & \mathit{Const} \\
 & | & \mathit{Ident} \\
 & | & \textbf{let}\ \mathit{Decl}\ \textbf{in}\ \mathit{Exp}\ \textbf{end} \\
 & | & \{[\![\mathit{Ident}\texttt{=}\mathit{Exp}]\!]^{*}_{,}\} \\
 & | & (\mathit{Exp}^{*}_{,}) \\
 & | & \texttt{\#}\mathit{Ident}\ \mathit{Exp} \\
 & | & \mathit{Exp}\ \mathit{Exp} \\
 & | & \mathit{Exp}\texttt{:}\mathit{TypeExp} \\
 & | & \mathit{Exp}\ \textbf{handle}\ \mathit{Match}^{+}_{|} \\
 & | & \textbf{raise}\ \mathit{Exp} \\
 & | & \textbf{case}\ \mathit{Exp}\ \textbf{of}\ \mathit{Match}^{+}_{|} \\
 & | & \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Exp}\ \textbf{else}\ \mathit{Exp}
\end{array}
$$

[2] Det finnes også spesielle likhetstypevariable som starter med to apostrofer, og imperative typevariable som starter med en eller to apostrofer og deretter en underscore.

```
          | fn Match+|
Match ::= Pattern => Exp
Decl  ::= val ⟦Pattern = Exp⟧+
          | fun ⟦Pattern = Exp⟧+and
          | type TypeParams? Ident = TypeExp
          | datatype TypeParams? Ident = TypeBind
          | exception Ident
          | Decl Decl
Pattern ::= _
          | Const
          | Ident
          | (Pattern*,)
          | Pattern Pattern
          | Pattern:TypeExp
TypeParams ::= 'Ident
          | ('Ident+,)
TypeBind ::= ⟦Ident⟦of TypeExp⟧?⟧+|
TypeExp ::= ⟦(TypeExp+,)⟧? Ident
          | TypeExp+*
          | {⟦Ident:TypeExp⟧+,}
          | TypeExp->TypeExp
          | (TypeExp)
```

Uttrykk

Notasjonen for funksjonsapplikasjon krever av og til at vi må legge på parenteser. For eksempel vil `f g x` bli tolket som `(f g) x`. Mener vi at resultatet av `g x` skal sendes med som parameter til `f` må vi skrive `f (g x)`.

Vilkårlige uttrykk kan dekoreres med typer, for eksempel `1 : int`. Fordi kolon binder svakere enn de fleste andre operatorer, er det en god vane og omslutte typing med parenteser. Vi skriver altså `(1 : int)`.

Case-uttrykk avsluttes *ikke* med et nøkkelord, som for eksempel **end**. Alternativer binder til nærmeste omsluttende case-uttrykk. Hvis hensikten er noe annet må ekstra parenteser legges på.

Deklarasjoner

Flere funksjonsdeklarasjoner kan kjedes sammen ved hjelp av det reserverte ordet **and**. Det benyttes for gjensidig rekursive funksjoner. I definisjonen [MTH90] kan dette gjøres med også andre deklarasjoner, som for eksempel datatype-deklarasjoner.

Typer

Typekonstruktøren for funksjonsrom `->` assosierer mot *høyre*. Det vil si at `int->int->int` blir tolket som `int->(int->int)`, altså som typen av funksjoner som tar et heltall og returnerer en funksjon fra heltall til heltall, og *ikke* som typen av funksjoner som tar en funksjon fra heltall til heltall og returnerer et heltall!

Moduler

Modulspråket i ML er basert på *strukturer*, *signaturer* og *funktorer*. Følgende analogi kan være en tankehjelp når man skal tilnærme seg disse begrepene:

- Strukturer tilsvarer Verdier
- Signaturer tilsvarer Typer
- Funktorer tilsvarer Funksjoner

Analogien holder ikke helt, men som en første intuisjon er det greit nok. Grammatikken under er en sterk forenkling av modulspråket i SML.

BNF

```
StructExp ::= struct Decl end
            | Ident [( [structure Ident = StructExp ]+ ) ]?
StructDecl::= structure Ident [: SigExpr ]?= StructExpr
            | StructDecl ; StructDecl
SigExp    ::= sig Spec end
            | Ident
SigDecl   ::= signature Ident = SigExp
Spec      ::= val Ident : TypeExp
            | type TypeParams? Ident
            | datatype TypeParams? Ident = TypeBind
            | exception Ident
            | Spec Spec
FunctorDecl::= functor
           Ident([structure Ident:SigExp]+):SigExp =
               StructExp
```

Strukturer

En struktur er en samling av deklarasjoner, for eksempel av typer og funksjoner. I det enkleste tilfellet omsluttes en slik samling av nøkkelordene **struct** og **end**. Alternativt kan vi definere en struktur ved å applisere en funktor (se under).

For deklarasjoner kan man i SML-NJ erstatte nøkkelordet **structure** med **abstraction**. For moduler som implementerer abstrakte datatyper, der det er viktig å beskytte den interne datastrukturen mot innsyn, er **abstraction** mer hensiktsmessig. Dette ligger dog utenfor definisjonen av Standard ML.

Signaturer

En signatur definerer typingen til de navnene man vil gjøre synlige i en gitt struktur, grensesnittet om man vil. Kroppen består av en sekvens av spesifikasjoner, av typer, funksjoner og unntak. Disse kan adskilles av semikolon, men det er ignorert i grammatikken.

En spesifikasjon på formen **type** `minType` angir en opak type hvis representasjon skal være ukjent utad.

Funktorer

En funktor er en "funksjon" fra strukturer til strukturer. Hver formelle struktur-parameter beskrankes med en signatur (analogt med typing av formelle variable). Legg merke til at listen av formelle parametre **ikke** skal adskilles med skilletegn av noe slag.

Topp-nivå dialogen

Språket som SML-NJ aksepterer på "topp-nivå" kan beskrives ved følgende grammatikk, der vi legger merke til at alle setninger avsluttes med semikolon:

```
Prog ::= Expr ;
       | Decl ;
       | StructExp ;
       | StructDecl ;
       | SigExp ;
       | SigDecl ;
       | Spec ;
       | FunctorDecl ;
```

Referanser

[Dah92] O.-J. Dahl. *Verifiable Programming*. International Series in Computer Science. Addison-Wesley, 1992.

[GJ87] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 1987.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, 1990.

[Oma94] K. Omang. ML - et typet, funksjonelt programmeringsspråk", og tilleggsstoff. Kompendium, Universitetet i Oslo, 1994. In Norwegian.

[Pau91] L. C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991.

[TAA94] L. Tafjord, K. Aasen, and D. Asheim. In211 - Oppgavekompendium med l sningsforslag og tilleggsstoff. Kompendium, Universitetet i Oslo, 1994.

[Wik87] Å. Wikstrøm. *Functional programming using Standard ML*. International Series in Computer Science. Prentice-Hall, 1987.