Løsningsforslag til eksamen i IN 211 høsten 2002

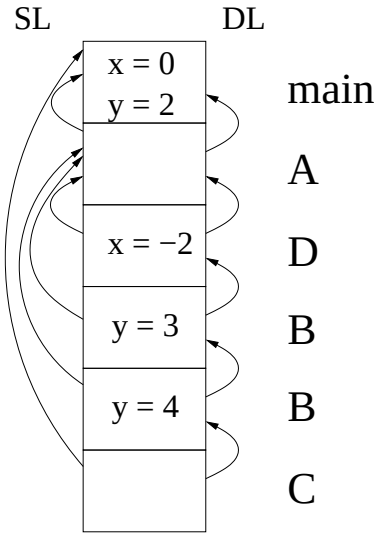
Ragnhild Kobro Runde og Gerhard Skagestein

4. desember 2002

Oppgave 1: Kjøresystemer

1a: Statisk skop

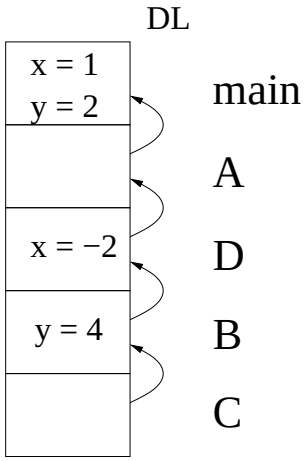
Print-setningen vil skrive ut 2, da det i C er main sin y som gjelder og denne ikke har fått endret sin verdi underveis.



1b: Dynamisk skop

1. Statisk link er ikke lenger nødvendig, siden den nettopp brukes for å finne ikke-lokale variable når vi har statisk skop. Dynamisk link er fortsatt nødvendig. Både for å kunne returnere til forrige aktiveringsblokk på stakken, og fordi det nå er denne vi vil følge for å finne ikke-lokale variable.

2. Print-setningen vil nå skrive ut 4, da C er kalt fra B og denne nettopp har satt sin y-verdi til å være 4.



1c: Parameteroverføring

1. Det enkleste her er kanskje å la kalleren og den kalte metoden ha variable med samme navn, for eksempel:

```
void main() {
  void P(int x) {
    int a = 2;
    print(x);
  }

  void Q {
    int a = 5;
    P(a);
  }

  Q();
}
```

   Ved call-by-name vil variabelen x som skrives ut i P være bundet til Q.a, og verdien 5 skrives ut. Ved call-by-text vil derimot x være bundet til den lokale P.a, og verdien 2 vil bli skrevet ut.

2. Den mest nærliggende løsningen er kanskje å ha en egen run-time rutine som evaluerer uttrykket hver gang det brukes — tilsvarende thunk-rutinen ved name-overføring (se læreboken side 101–102).

   En alternativ løsning vil være å se på call-by-text som en makro, der kompilatoren starter med å tekstlig substituere alle parametrene. Ved ulike kall på samme metode, vil vi da få substitusjon til flere ulike metoder. Senere variabel-aksess vil da gå akkurat som normalt. Dette vil også si at kompilatoren kan gjøre statisk sjekking av disse parametrene på samme måte som for vanlige variable.

Oppgave 2: ML-map

2a: Liste-implementasjon av map

```
structure Map_impl: Map_def =
struct
  type (''key, ''val) Map = (''key * ''val) list;
  exception noValue;
  val empty = [];

  fun size(m) = case m of [] => 0
                | x :: r     => 1 + size(r);

  fun containsKey(m, k) = case m of [] => false
                          | (x,v) :: r => x = k orelse containsKey(r,k);

  fun containsValue(m, v) = case m of [] => false
                            | (k,x) :: r => x = v orelse containsValue(r,v);

  fun get(m, k) = case m of [] => raise noValue
                  | (x,v) :: r => if x = k then v else get(r,k);

  fun put(m,k,v) = case m of [] => [(k,v)]
                   | (k',v')::r => if k = k' then (k,v) :: r
                                   else (k',v') :: put(r,k,v);

  fun remove(m,k) = case m of [] => []
                    | (x,v) :: r => if k = x then r
                                    else (x,v)::remove(r,k);

  fun union(m1,m2) = case m2 of [] => m1
                     | (k,v) :: r  => put(union(m1,r),k,v);
end;
```

2b: Bruk av map

Dette kan selvsagt gjøres på flere måter. Mitt forslag er basert på følgende ide:

1. Bruk `values` for å få ut en liste med lister av de som har fått besøk.
2. Hjelpefunksjonen `flatut` konverterer denne til en enkel liste med samme verdier.
3. Funksjonen `konverter` går gjennom den utflatede listen. For hvert navn `x` brukes hjelpefunksjonen `antall` til å telle opp hvor mange denne har fått besøk av. Hvert slikt (navn,antall)-par settes så inn i det konverterte map'et.

Merk at alle som har fått besøk mer enn en gang vil puttes inn i map'et flere ganger, først med verdien 1, deretter med verdien 2 og så videre. En mer effektiv løsning kunne brukt let-konstruksjonen og latt antall også returnere listen med alle forekomster av navnet `x` fjernet.

```
fun flatut(l) = case l of [] => []
                | x :: r => x @ flatut(r);

fun antall(x,l) = case l of [] => 0
                  | y :: r     => if x = y then 1 + antall(x,r)
                                  else antall(x,r);

fun konverter(l) = case l of [] => empty
                   | x :: r     => put(konverter(r), x, antall(x,r) + 1);

fun antallbesok(besoktmap) = konverter(flatut(values(besoktmap)));
```

2c: Map implementert som funksjonsrom

```
type (''key, ''val) Map = ''key -> ''val;
```

- De eneste Map-funksjonene som ikke kan implementeres med funksjonsrom er funksjonene `size` og `containsValue`, siden disse krever at vi kan løpe gjennom hele strukturen. Dette er som kjent ikke mulig med funksjonsrom.

  (Det er heller ikke mulig å lage funksjonene `keys` og `values` som nevnt i oppgave 2b, men dette var det ikke spurt om her.)

- Implementasjon av resten av funksjonene:

  ```
  val empty = fn x => raise noValue;

  fun containsKey(m, k) = m(k)=m(k) handle noValue => false;

  fun get(m,k) = m(k);

  fun put(m,k,v) = fn x => if x = k then v else m(x);

  fun remove(m,k) = fn x => if x = k then raise noValue else m(k);

  fun union(m1,m2) = fn x => m2(x) handle noValue => m1(x);
  ```

  Forklaring:

  `containsKey` skal returnere `false` hvis nøkkelen ikke finnes. Siden oppslag med ikke-eksisterende nøkkel gir unntaket `noValue` kan vi enkelt fange opp dette og gi `false` istedenfor. Problemet er hvordan returnere `true`. Dette har jeg her løst ved å teste om `m(x)` er lik seg selv. Hvis nøkkelen finnes vil dette alltid slå til, finnes ikke nøkkelen får vi unntaket som nevnt over.

  I `union(m1,m2)` skal mappingene i `m2` gis prioritet. Dette gjøres her ved å slå opp i denne først, og så slå opp i `m1` kun hvis `m2` gir unntak.

Oppgave 3: Hypervektorer: Grammatikk

3a: Typing

1. Vi går ut fra at språket ikke har eksplisitte deklarasjoner — slike er i alle fall ikke vist i eksemplene og i grammatikken gitt i oppgaven. Da er den eneste muligheten strukturell kompatibilitet — se læreboka side 140.

2. Operatoren virker både som et hypervektor skalarprodukt og som et heltalls (og antagelig også flytende talls) skalarprodukt, avhengig av operandene. Altså har vi operator-overlasting (som er en ad-hoc-polymorfisme, se læreboka side 147, figur 3.10)

3. Et språk er sterkt typet dersom kompilatoren kan garantere at utførelsen ikke kan gi typefeil (se læreboka side 138). Dette er åpenbart ikke tilfelle, siden grammatikken tillater $[3, 5] \bullet 7$.

   I forelesningene er gitt en noe videre definisjon: "Et programmeringsspråk er sterkt typet dersom alle typefeil blir diagnostisert. Hvis alle bindinger er statiske, kan mesteparten av typesjekkingen også gjøres statisk. Hvis bindingen er dynamisk, må typesjekkingen også være dynamisk."

   Språket kan ansees som sterkt typet i henhold til den videre definisjonen (dvs. at det er typesikkert), siden korrekt typing kan sjekkes ved hjelp av typebeskrivelser i operandene under utførelsen. Oppgaveteksten spesifiserer ikke semantikken i bruk av skalarmultiplikasjonsoperatoren mellom vektorer av ulik lengde. Det er nærliggende å tolke manglende elementer som 0 eller en tom vektor, men dersom dette ikke gjøres, vil vi få en kjøretidsfeil. Om dette er en typefeil, er avhengig av om vektorlengden anses som en del av typen eller ikke.

3b: LL(1)

1. LL(1)-grammatikk:

   $\langle prog \rangle \rightarrow \langle assign \rangle\ \langle assignlist \rangle$
   $\langle assignlist \rangle \rightarrow \langle assign \rangle\ \langle assignlist \rangle \mid \varepsilon$
   $\langle assign \rangle \rightarrow \textbf{var} \leftarrow \langle expr \rangle\ \textbf{;}$
   $\langle expr \rangle \rightarrow \textbf{var}\ \langle factor \rangle \mid \textbf{[}\ \langle expr \rangle\ \langle exprlist \rangle\ \textbf{]}\ \langle factor \rangle \mid \textbf{tall}\ \langle factor \rangle$
   $\langle factor \rangle \rightarrow \bullet\ \langle expr \rangle \mid \varepsilon$
   $\langle exprlist \rangle \rightarrow \textbf{,}\ \langle expr \rangle\ \langle exprlist \rangle \mid \varepsilon$

2. Bevis for LL(1):

	MTT	**Startmengde**	**Etterfølgermengde**
$\langle prog \rangle$	Nei	var	
$\langle assignlist \rangle$	Ja	var	
$\langle assign \rangle$	Nei	var	var
$\langle expr \rangle$	Nei	var [tall	;] ,
$\langle factor \rangle$	Ja	•	;] ,
$\langle exprlist \rangle$	Ja	,	]

Produksjon	Utvidet startmengde	Disjunkt?
$\langle prog \rangle \rightarrow \langle assign \rangle\ \langle assignlist \rangle$	var	JA
$\langle assignlist \rangle \rightarrow \langle assign \rangle\ \langle assignlist \rangle$ $\langle assignlist \rangle \rightarrow \varepsilon$	var (tom)	JA
$\langle assign \rangle \rightarrow$ var $\leftarrow \langle expr \rangle$;	var	JA
$\langle expr \rangle \rightarrow$ var $\langle factor \rangle$ $\langle expr \rangle \rightarrow$ [$\langle expr \rangle\ \langle exprlist \rangle$] $\langle factor \rangle$ $\langle expr \rangle \rightarrow$ tall $\langle factor \rangle$	var [tall	JA
$\langle factor \rangle \rightarrow \bullet\ \langle expr \rangle$ $\langle factor \rangle \rightarrow \varepsilon$	• ;] ,	JA
$\langle exprlist \rangle \rightarrow$, $\langle expr \rangle\ \langle exprlist \rangle$ $\langle exprlist \rangle \rightarrow \varepsilon$	,]	JA

Alle utvidede startmengder er disjunkte, altså er grammatikken LL(1).

3c: Regulære uttrykk

1. Regulært uttrykk (merk at [og] er grunnsymboler, mens (og) er metaparenteser):

$$[\ (tall \mid var)\ (\bullet\ (tall \mid var))^*\ (,\ (tall \mid var)\ (\bullet\ (tall \mid var))^*)^*\ ]$$

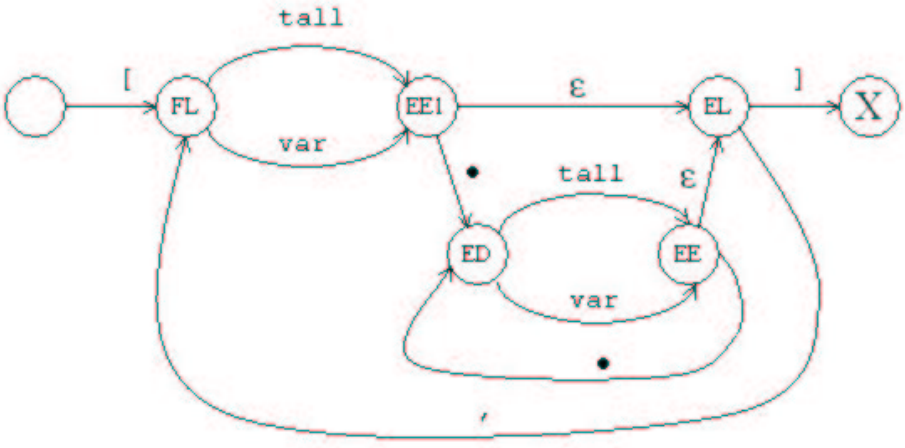
Regulær grammatikk (basert på den ikke-deterministiske automaten i neste punkt):

$\langle UTTRYKK \rangle \rightarrow$ **[** $\langle FL \rangle$
$\langle FL \rangle \rightarrow$ **tall** $\langle EE1 \rangle$ | **var** $\langle EE1 \rangle$
$\langle EE1 \rangle \rightarrow \bullet\ \langle ED \rangle \mid \langle EL \rangle$
$\langle ED \rangle \rightarrow$ **tall** $\langle EE \rangle$ | **var** $\langle EE \rangle$
$\langle EE \rangle \rightarrow \bullet\ \langle ED \rangle$ | $\langle EL \rangle$
$\langle EL \rangle \rightarrow$ **,** $\langle FL \rangle$ | **]**

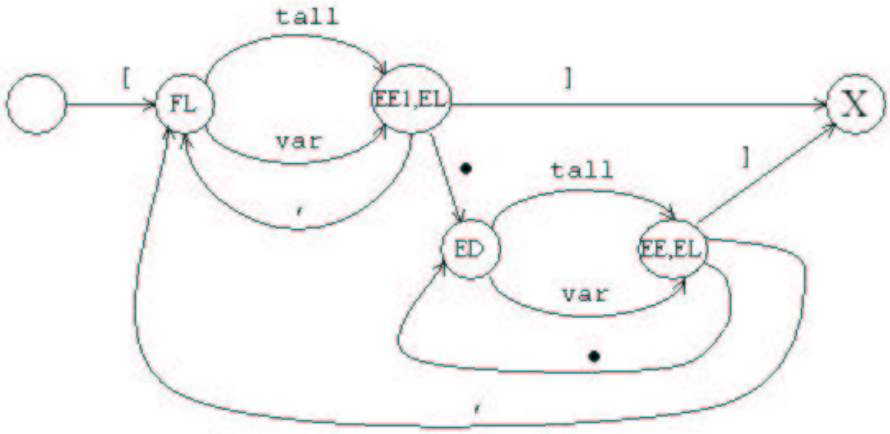
Betydningen av metasymbolene er som følger:

FL	Før liste
EE1	Etter første element
EE	Etter element
ED	Etter •
EL	Etter liste

2. Ikke-deterministisk automat:



3. Deterministisk automat:

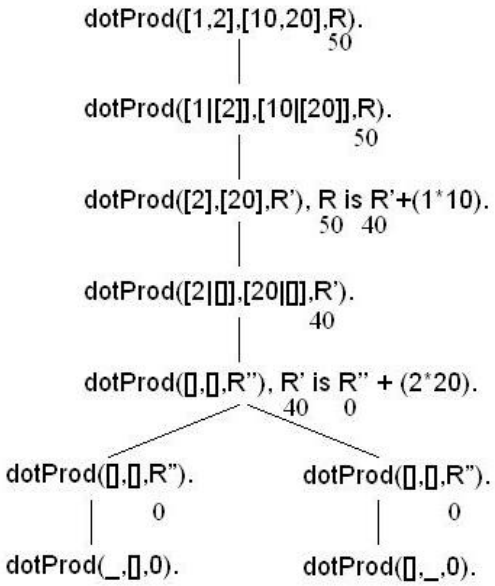


Denne automaten kan forenkles videre ved å slå sammen FL og ED, samt (EE1,EL) med (EE,EL).

Oppgave 4: Hypervektorer: Prolog

4a: Spørringer

1. Søketre:



2. Det finnes en løsning til. Den ene løsningen baserer seg på regelen `dotProd(_,[],0)`, den andre på `dotProd([],_,0)`. I begge tilfellene bindes verdien 50 til R.

3. Cut-varianten:

```
dotProd([],_,0).
dotProd(_,[],0).
dotProd([H1|T1],[H2|T2],Result) :- dotProd(T1,T2,R),
                                   Result is R + (H1*H2),!.
```

Uten prosedurelle aspekter:

```
dotProd([],[],0).
dotProd([],L,0):- L \==[].
dotProd(L,[],0):- L \==[].
dotProd([H1|T1],[H2|T2],Result) :- dotProd(T1,T2,R),
                                   Result is R + (H1*H2).
```

4. Intuitivt kan man hevde at `dotProd([1,2] ,X, 50)` vil kunne gi uendelig mange løsninger (for eksempel X=[10,20] og X=[30,10]), og at spørringen derfor er uten mening. Men det egentlige problemet er at is-predikatet ikke unifiserer på vanlig måte. Vi kan i uttrykket `Result is R + (H1*H2)` ikke binde en verdi til `Result` og forvente at Prolog binder verdier til R, H1 og H2.

4b: Hypervektorer

Det går an å tenke på to måter: Enten å beregne skalarproduktverdiene for alle vektorene og deretter summere skalarene, eller å akkumulere opp skalarproduktverdiene for alle vektorene etter hvert i samme akkumulator. Den følgende løsningen bygger på den siste tenkemåten.

```
hyperDotProd([],[],0).
hyperDotProd([],L,0):- L \==[].
hyperDotProd(L,[],0):- L \==[].

hyperDotProd([H1|T1],[H2|T2],Result) :- hyperDotProd(H1,H2,ResA),
                                        hyperDotProd(T1,T2,ResB),
                                        Result is ResA + ResB.
hyperDotProd(H1, H2, Result) :- number(H1),
                                number(H2),
                                Result is H1 * H2.
```

Oppgave 5: Hypervektorer: ML

1. ```
   type hyperelement = tall of int | liste of hyperelement list;
   type hypervektor = hyperelement list;
   ```

2. `((1,2),(3,4,5))` vil med representasjonen over skrives som:
   `[liste([tall(1),tall(2)]), liste([tall(3),tall(4),tall(5)])]`

   `((10,20),(30,40,50))` vil tilsvarende bli skrevet som:
   `[liste([tall(10),tall(20)]), liste([tall(30),tall(40),tall(50)])];`

3. Min versjon av `hyperdotprod` takler tomme lister, ved at disse tolkes som en null-vektor (som forklart i innledningen om hypervektorer). Den takler også lister av ulik lengde, ved å fylle ut med null-elementer der det er nødvendig. Oppgaveteksten legger opp til at tomme lister skal takles, men ikke nødvendigvis lister av ulik lengde. I praksis vil begge deler føre til at produktet alltid blir null.
   ```

Det å ta produktet av et tall og en liste er derimot ikke definert, og gir her unntaket syntaksfeil.

```
exception syntaksfeil;

fun hyperdotprod(v1:hypervektor, v2:hypervektor):int =
case v1 of [] => ( case v2 of [] => 0
                   | y :: s      => 0 )
| x :: r      => ( case v2 of [] => 0
                   | y :: s      =>

(* Hovedtilfellet. v1 er listen x::r, mens v2 er listen y::s *)

   case (x,y) of (tall(i),tall(j)) => i*j + hyperdotprod(r,s)
   | (liste(l),liste(k))           => hyperdotprod(l,k) + hyperdotprod(r,s)
   | _                             => raise syntaksfeil );
```

4. Det er ikke mulig å erstatte typen `hypervektor` med `'a list` av to grunner:

   For det første kan vi ha ulik nesting innenfor samme hypervektor. For eksempel vil hypervektoren `[[1,2],[[3,4,5]]]` være en liste bestående av en `int list` og en `int list list`. Dette er ikke mulig i ML, heller ikke med typen `'a list`.

   For det andre trenger vi i funksjonen `hyperdotprod` å vite om elementene vi skal ta produktet av er tall eller lister, for å vite om vi skal bruke vanlig multiplikasjon eller et rekursivt kall på `hyperdotprod`.

Oppgave 6: Hypervektorer: Språkvalg

Det viktigste her er at kriteriene er fornuftige, og at det er konsistens mellom kriteriene, språkvalget og begrunnelsen. Siden det at man behersker et språk ofte er den viktigste begrunnelsen for språkvalget, har jeg tatt dette kriteriet med i mitt forslag til kriterieliste:

- Jeg kan språket
- Språket må tillate rekursjon
- Språket må tillate overlasting av operatorer og dynamisk binding
- Det er lite bruk for tilordning (assignment)
- Lite behov for eksplisitte løkkekonstruksjoner — bedre om iterasjonene er implisitte

. . . hvilket peker i retning av et funksjonelt språk, for eksempel ML!