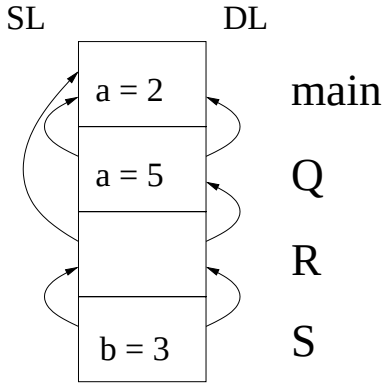
Løsningsforslag til eksamen i IN 211 høsten 2001

Ragnhild Kobro Runde

6. desember 2001

Oppgave 1: Kjøresystemer

1a: Statisk og dynamisk link



1b: Parameteroverføring

1. Verdioverføring: `a` endres ikke av `P`, og det som skrives ut er `2`.

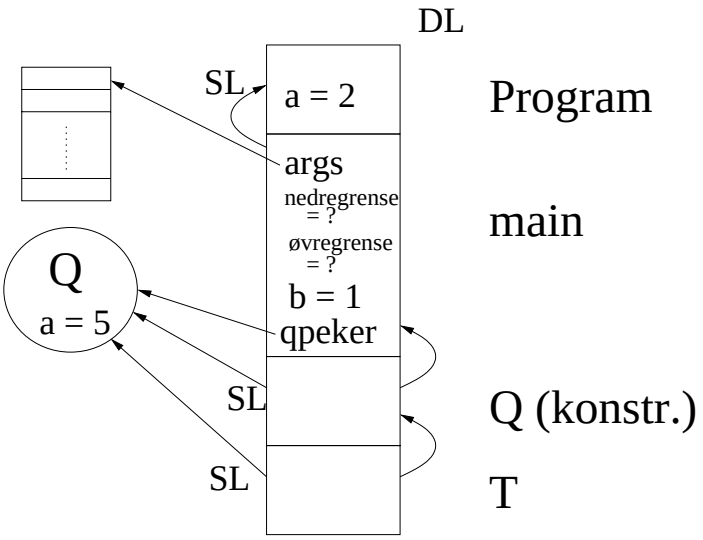
   Name-overføring: Det som blir utført av kallet på `P` er setningene `a = a + 1` og `a = a + (a + b)`, og verdien `9` vil bli skrevet ut.

2. Referanseoverføring: `x` blir nå en peker til `b` i `S`, mens `y` og `z` peker til `a` i `R`. Setningen `y = y + 1` fører dermed til at `a` settes til `3`, mens setningen `z = z + x` fører til at `a` settes til `6`, og det er denne verdien for `a` som vil bli skrevet ut.

   Verdi-resultat: Ved metodekall kopieres verdiene av de aktuelle parameterne, slik at `x` får verdien `3`, mens `y` og `z` får verdien `2`. Rett før `P` returnerer, vil `x` ha verdien `3`, `y` ha verdien `3` og `z` ha verdien `5`. Siden vi også har resultatoverføring vil også verdiene bli kopiert tilbake. Hvis vi antar at dette gjøres fra venstre mot høyre, får vi da at `b` får verdien `3`, mens `a` først får verdien `3` (fra `y`), og deretter verdien `5` (fra `z`). Det som skrives ut av print-setningen blir da `5`.

3. Hvis vi har referanseoverføring er problemet at vi ikke har noen lokasjon for `a + b` som `x` kan peke til. Tilsvarende har vi for resultat-overføring ikke noen lokasjon å kopiere `x` tilbake til. En mulig løsning på begge disse problemene kunne vært å opprette en midlertidig lokasjon i kallerens (dvs. `S`) omgivelse.

1c: Objekter



Oppgave 2: Prolog-grammatikk

2a: Syntaksmengder

Metasymbol	MTT	Startmengde	Etterfølgermengde
⟨*statmt*⟩		name :-	
⟨*fact*⟩		name	
⟨*pred*⟩		name	. :-
⟨*rule*⟩		name :-	
⟨*arglist*⟩		arg [	)
⟨*list*⟩		[	)]
⟨*listelems*⟩	JA	arg	]

2b: Ikke LL(1)

Grammatikken er ikke LL(1) av fire grunner (her var det nok å gi *ett* moteksempel):

- ⟨*statmt*⟩ er enten ⟨*fact*⟩ eller ⟨*rule*⟩, og begge disse kan starte med **name**.
- ⟨*arglist*⟩ har to alternative produksjoner som begge starter med **arg**.
- ⟨*listelems*⟩ har tre alternative produksjoner som alle starter med **arg**.
- Definisjonen av ⟨*pred*⟩ skjuler egentlig to produksjoner:

  ⟨*pred*⟩ → **`name`**
  ⟨*pred*⟩ → **`name (`** ⟨*arglist*⟩ **`)`**

  Begge disse starter med **name**.

2c: LL(1)

Forslag til ny grammatikk, der ⟨*list*⟩ er som før:[1].

⟨*statmt*⟩ → ⟨*pred*⟩ ⟨*statmtrest*⟩ | ⟨*rulerest*⟩
⟨*statmtrest*⟩ → ⟨*factrest*⟩ | ⟨*rulerest*⟩
⟨*factrest*⟩ → `.`
⟨*rulerest*⟩ → `:- body .`
⟨*pred*⟩ → `name` ⟨*predrest*⟩
⟨*predrest*⟩ → *ε* | `(` ⟨*arglist*⟩ `)`
⟨*arglist*⟩ → `arg` ⟨*arglistrest*⟩ | ⟨*list*⟩
⟨*arglistrest*⟩ → *ε* | `,` ⟨*arglist*⟩
⟨*list*⟩ → `[` ⟨*listelems*⟩ `]`
⟨*listelems*⟩ → *ε* | `arg` ⟨*listelemsrest*⟩
⟨*listelemsrest*⟩ → *ε* | `I` ⟨*listelemsrest2*⟩
⟨*listelemsrest2*⟩ → ⟨*list*⟩ | `var`

Metasymbol	MTT	Startmengde	Etterfølgermengde
⟨*statmt*⟩		name :-	
⟨*statmtrest*⟩		. :-	
⟨*factrest*⟩		.	
⟨*rulerest*⟩		:-	
⟨*pred*⟩		name	. :-
⟨*predrest*⟩	JA	(	. :-
⟨*arglist*⟩	JA	arg [	)
⟨*arglistrest*⟩		,	)
⟨*list*⟩		[	)]
⟨*listelems*⟩	JA	arg	]
⟨*listelemrest*⟩	JA	I	]
⟨*listelemrest2*⟩		[var	]

Må beregne de utvidede startmengdene for metasymboler med flere mulige produksjoner:

Produksjon	Utvidet startmengde	Disjunkt
⟨*statmt*⟩ → ⟨*pred*⟩ ⟨*statmtrest*⟩	name	
⟨*statmt*⟩ → ⟨*rulerest*⟩	:-	JA
⟨*statmtrest*⟩ → ⟨*factrest*⟩	.	
⟨*rulerest*⟩ → ⟨*rulerest*⟩	:-	JA
⟨*predrest*⟩ → *ε*	. :-	
⟨*predrest*⟩ → (⟨*arglist*⟩)	(	JA
⟨*arglistrest*⟩ → *ε*	)	
⟨*arglistrest*⟩ → , ⟨*arglist*⟩	,	JA
⟨*listelems*⟩ → *ε*	]	
⟨*listelems*⟩ → arg ⟨*listelemsrest*⟩	arg	JA
⟨*listelemsrest*⟩ → *ε*	]	
⟨*listelemsrest*⟩ → I ⟨*listelemsrest2*⟩	I	JA
⟨*listelemsrest2*⟩ → ⟨*list*⟩	[	
⟨*listelemsrest2*⟩ → var	var	JA

Altså er grammatikken LL(1).

2d: Recursive descent

Her godtas både Java og pseudokode.

Vi bruker følgende hjelpemetoder:

lesSymbol(String s) sjekker om gjeldende symbol er lik s. Leser også inn neste symbol.

[1] Denne kan gjøres enklere, ved f.eks. å ta med **name** allerede i det første alternativet til ⟨*statmt*⟩

sjekkSymbol(String s) sjekker om gjeldende symbol er lik s (uten å lese inn neste symbol).

Disse hjelpemetodene skriver også ut en eventuell feilmelding hvis symbolet ikke stemmer.

```
void parseList() {
    lesSymbol("[");
    parseListelems();
    lesSymbol("]");
}

void parseListelems() {
    if (sjekkSymbol("]")) {
        // Tom høyreside
    } else if (sjekkSymbol("arg")) {
        lesSymbol("arg");
        parseListelemsrest();
    }
}

void parseListelemsrest() {
    if (sjekkSymbol("]")) {
        // Tom høyreside
    } else if (sjekkSymbol("I")) {
        lesSymbol("I");
        parseListelemsrest2();
    }
}

void parseListelemsrest2() {
    if (sjekkSymbol("[")) {
        parselist();
    } else if (sjekkSymbol("var")) {
        lesSymbol("var");
    }
}
```

Oppgave3: Regulære uttrykk

3a: Regulært uttrykk

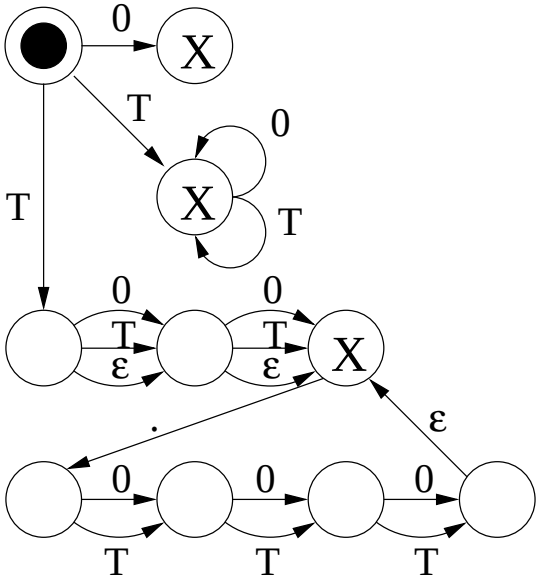
Kan dele opp det regulære uttrykket i tre deler:

- Beløp på 0 kroner: 0
- Beløp uten gruppering: $T(0|T)^*$
- Beløp med gruppering med punktum: $T(0|T)^?(0|T)^?(.(0|T)(0|T)(0|T))^*$

Samlet regulært uttrykk blir da: $0|(T(0|T)^*)|(T(0|T)^?(0|T)^?(.(0|T)(0|T)(0|T))^*)$
(Dette kan forenkles noe, siden beløp med og uten gruppering overlapper for alle beløp under tusen kroner.)

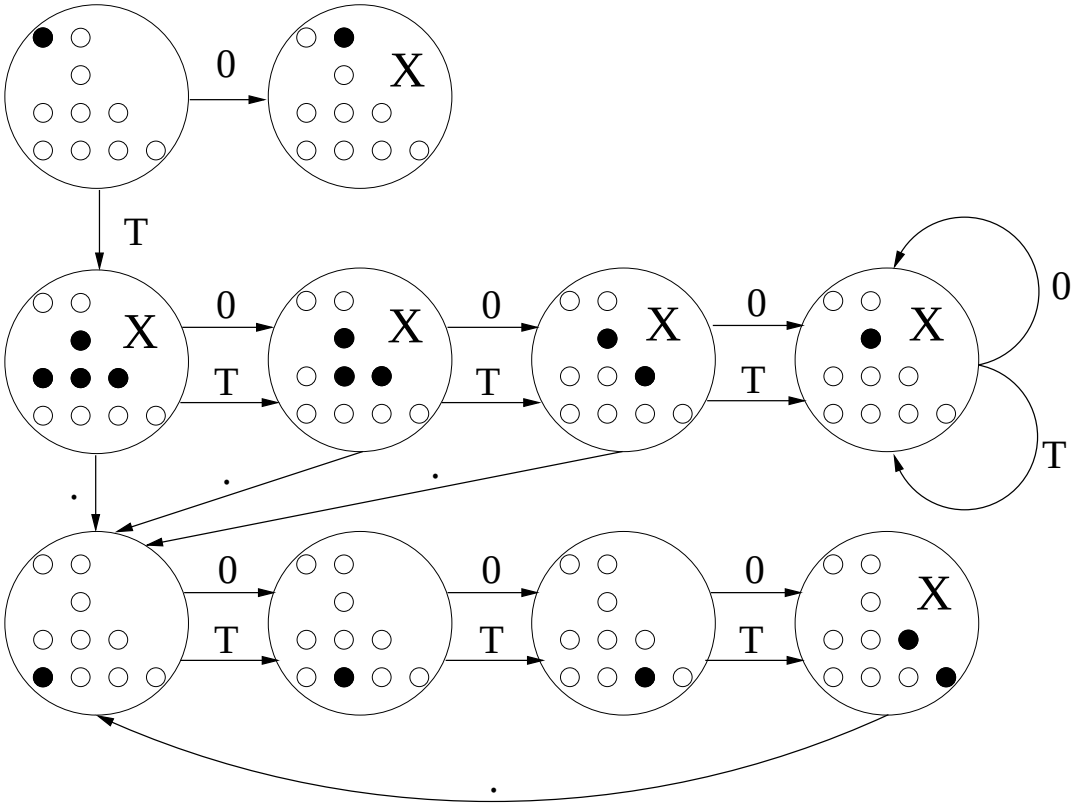
3b: Ikke-deterministisk automat

En mulig løsning (satt opp relativt rett frem etter det regulære uttrykket i 3a):



3c: Deterministisk automat

Overgang fra den ikke-deterministiske automaten i forrige punkt gir:



Oppgave 4: Bussruter i ML

```
type bussrute = int * string list;
type rutehefte = bussrute list;
```

4a: Funksjonen steder

```
fun steder(nr:int, rh:rutehefte):string list =
case rh of [] => []
| (n,s) :: r  => if n=nr then s
                 else steder(nr,r);
```

4b: Funksjonen busser

Trenger en hjelpefunksjon `has` som sjekker om en buss er med i en liste av stoppesteder. Denne kan skrives på vanlig måte:

```
fun has(l,x) =
case l of [] => false
| y :: r     => x = y orelse has(r,x);
```

Funksjonen `busser` kan de se slik ut:

```
fun busser(st:string, rh:rutehefte):int list =
case rh of [] => []
| (n,s) :: r  => if has(s,st) then n :: busser(st,r)
                 else busser(st,r);
```

4c: Ny bussrute

```
fun leggtil(nr:int, ss:string list, rh:rutehefte):rutehefte =
case rh of [] => [(nr,ss)]
| (n,s) :: r  => if nr = n then (nr,ss) :: r
                 else (n,s) :: leggtil(nr,ss,r);
```

4d: Nedlegging av stoppested

Det kan her være greit å ha en hjelpefunksjon som sletter et element fra en liste:

```
fun slett(l,x) =
case l of [] => []
| y :: r     => if x = y then r
                else y :: slett(r,x);
```

Her var det uspesifisert hva man skulle gjøre hvis en rute nå bare får ett eller ingen stoppesteder. Det tillates derfor flere løsinger her, også å ikke gjøre noe spesielt, som:

```
fun fjern(st:string, rh:rutehefte) =
case rh of [] => []
| (n,s) :: r  => (n,slett(s,st)) :: fjern(st,r);
```

4e: Bytte mellom busser

Her er det flere muligheter. Et alternativ er rett og slett å lage en liste med alle stoppestedene til busser som stopper i A, en annen liste med alle stoppestedene til busser som stopper i B, og så se om disse har minst ett felles stoppested.

```
fun finnalle(b:int list, rh:rutehefte):string list =
case b of [] => []
| i :: r     => steder(i,rh) @ finnalle(r,rh);


fun ikkeTomtSnitt(l1,l2):bool =
case l1 of [] => false
| x :: r      => has(l2,x) orelse ikkeTomtSnitt(r,l2);

fun maksEttBytte(A:string, B:string, rh:rutehefte):bool =
ikkeTomtSnitt(finnalle(busser(A,rh),rh), finnalle(busser(B,rh),rh));
```

4f: Implementasjon ved funksjonsrom

```
type rutehefte = int -> string list;
```

1. ```
   fun steder(nr:int, rh:rutehefte):string list = rh(nr);

   fun leggtil(nr:int, ss:string list, rh:rutehefte):rutehefte =
   fn(i) => if i = nr then ss else rh(i);

   fun fjern(st:string, rh:rutehefte):rutehefte =
   fn(i) => slett(rh(i),st);
   ```

   der `slett` er som i oppgave 4d.

2. Vi kan ikke lage funksjonen `busser` siden vi ikke har noen mulighet for å søke gjennom et funksjonsrom.

Busser i Prolog

5a: Oppslag

Spørsmålet

```
stopper(X,helsfyr).
```

vil gi alle rutene som stopper ved “helsfyr” bundet til `X` en etter en.

5b: Knutepunkter

Regel:

```
knutepunkt(A) :- stopper(X,A), stopper(Y,A), X \== Y.
```

Spørsmål:

```
knutepunkt(X).
```

Med faktaene over får vi: `nygård`, `teisen`, `helsfyr`, `teisen`, `helsfyr`, `nygård` (i denne rekkefølgen).
```

5c: Reiserute

En mulighet er å lage en regel som angir en reise uten bytter, og en regel som angir en reise med nøyaktig ett bytte:

```
ingenbytter(A,B,X) :- stopper(X,A), stopper(X,B), A \== B.

ettbytte(A,B,X,Y) :- stopper(X,A), stopper(X,C), A \== C,
                     stopper(Y,C), X \== Y,
                     stopper(Y,B), A \== B, B \== C.
```

Gitt to konkrete stoppesteder `a` og `b`, kan man da spørre:

```
ingenbytter(a,b,X); ettbytte(a,b,X,Y).
```

En enklere løsning er å ha alt i samme regel:

```
ingenEllerEttBytte(A,B,X,Y) :- stopper(X,A), stopper(X,C),
                               stopper(Y,C), stopper(Y,B).
```

Her tillates det at X og Y er samme buss, noe som da betyr at det ikke er nødvendig med noe bytte.