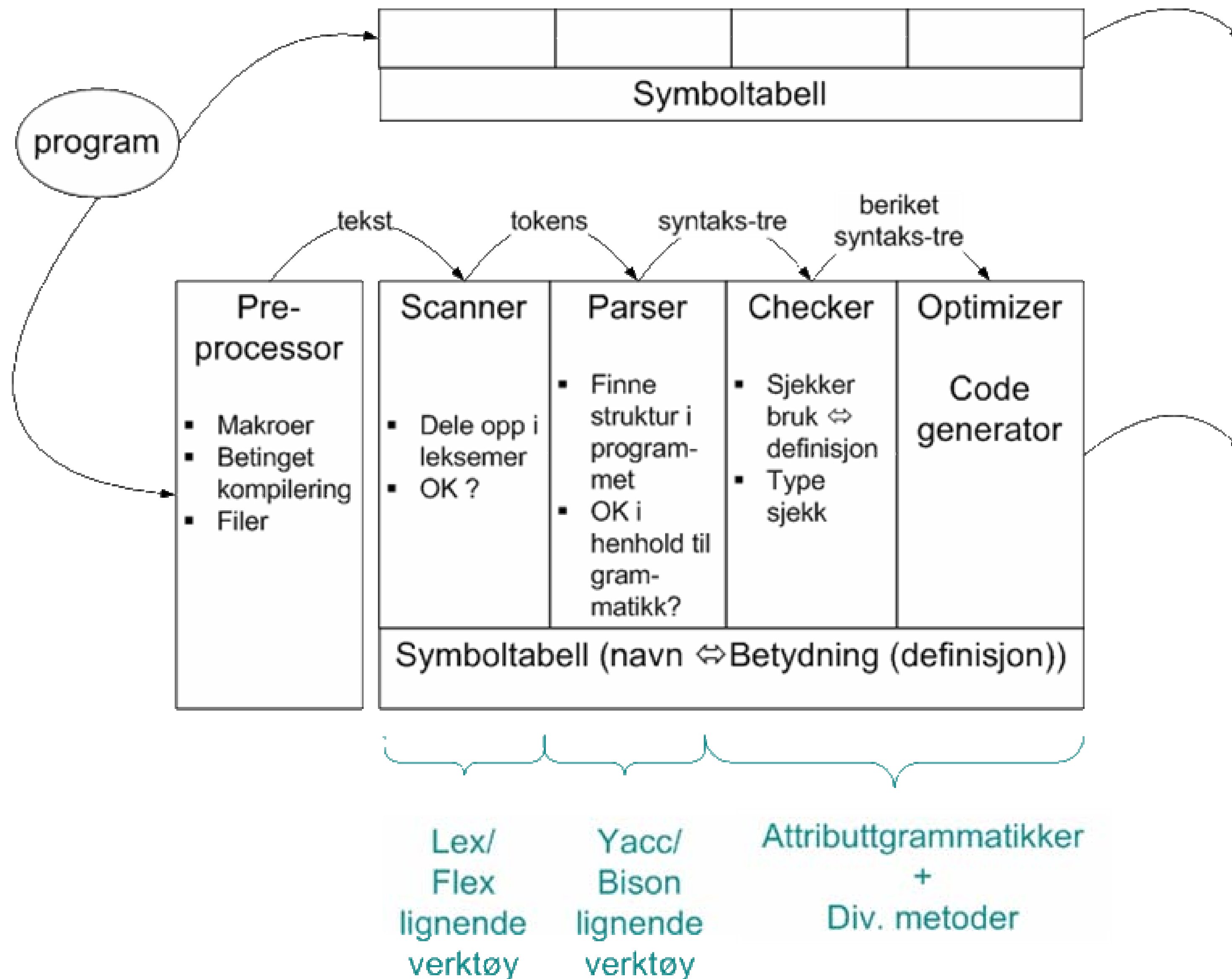
Anatomien til en kompilator - I

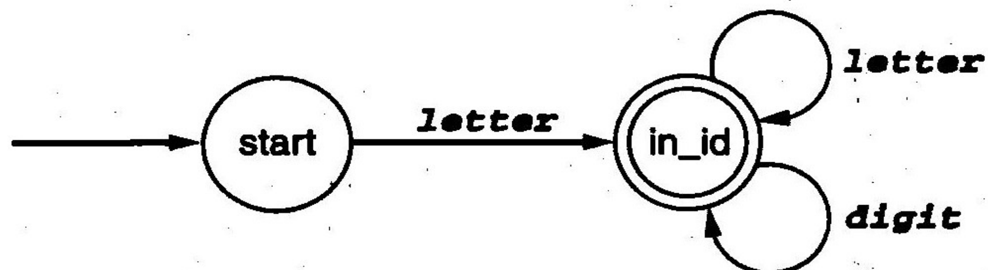


Framgangsmåte

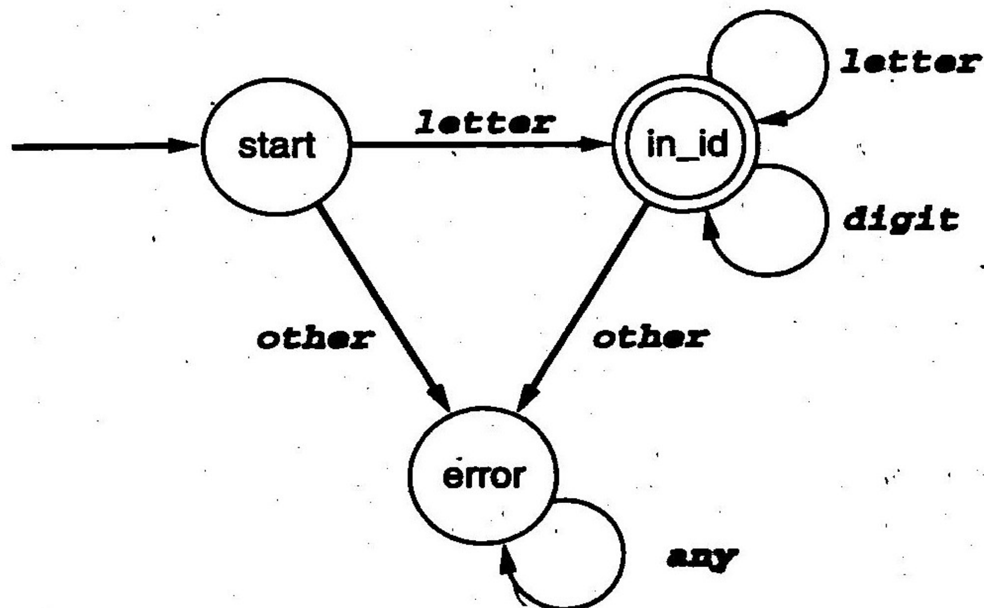
- for automatisk å lage en scanner

- Beskriv de forskjellige token-klassene som ”regulære uttrykk”
  - Eller litt mer fleksibelt, som ”regulære definisjoner”
- Omarbeid dette til en NFA (non-deterministic finite automaton)
  - Er veldig rett fram
- Omarbeid dette til en DFA (deterministic finite automaton)
  - Dette kan gjøres med en kjent, grei algoritme
- En DFA kan uten videre gjøres om til et program

DFA -1



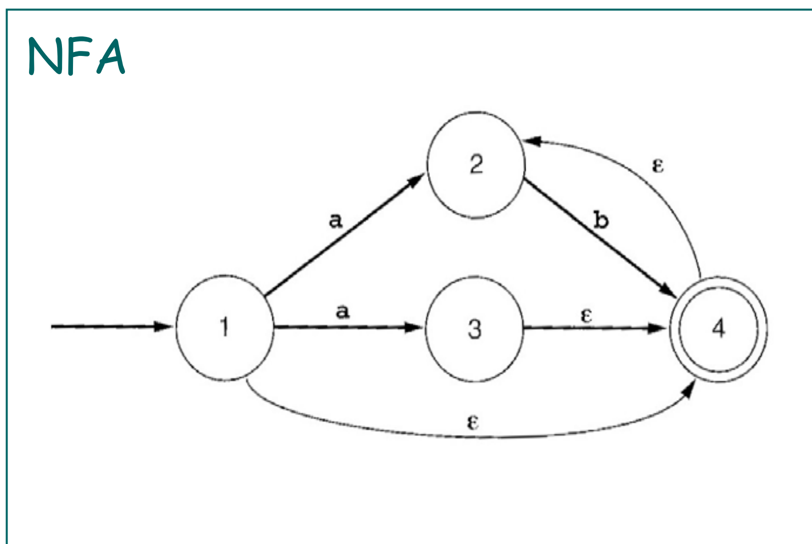
Funksjonen T: $S \times \Sigma$ -> S er ikke fullstendig definert

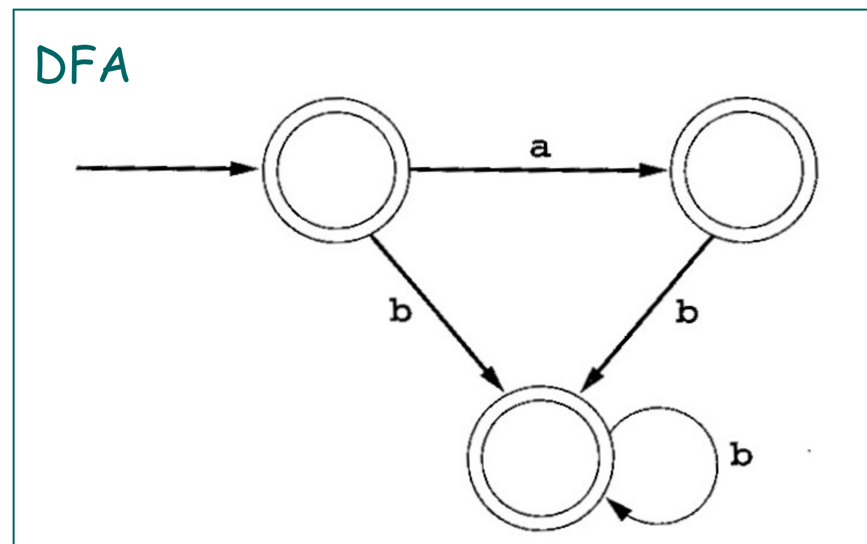


Denne utvidelsen (med en feiltilstand) er underforstått

NFA

- Kan ofte være lett å sette opp, spesielt ut fra et regulært uttrykk
- Kan ses på som syntaks-diagrammer



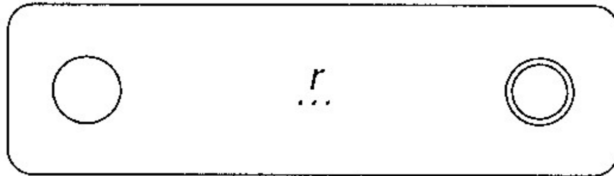


- ikke greit å gjøre til algoritme
- beskriver samme språk

INF 5110 - 2006

Thomson-konstruksjon I

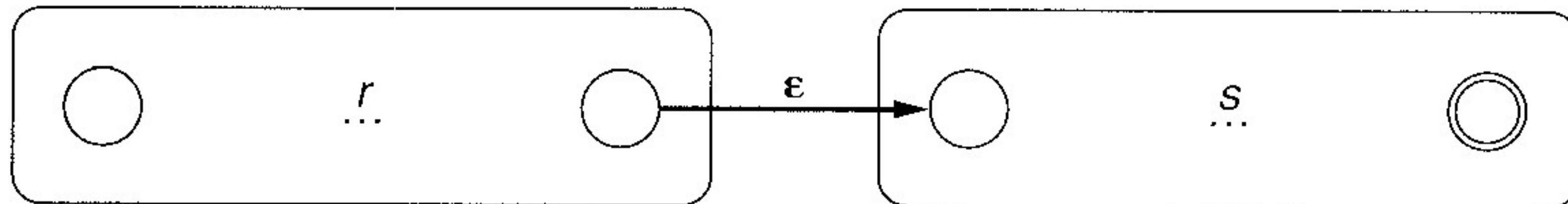
(Ethvert regulært uttrykk skal bli automat på denne formen:



a:

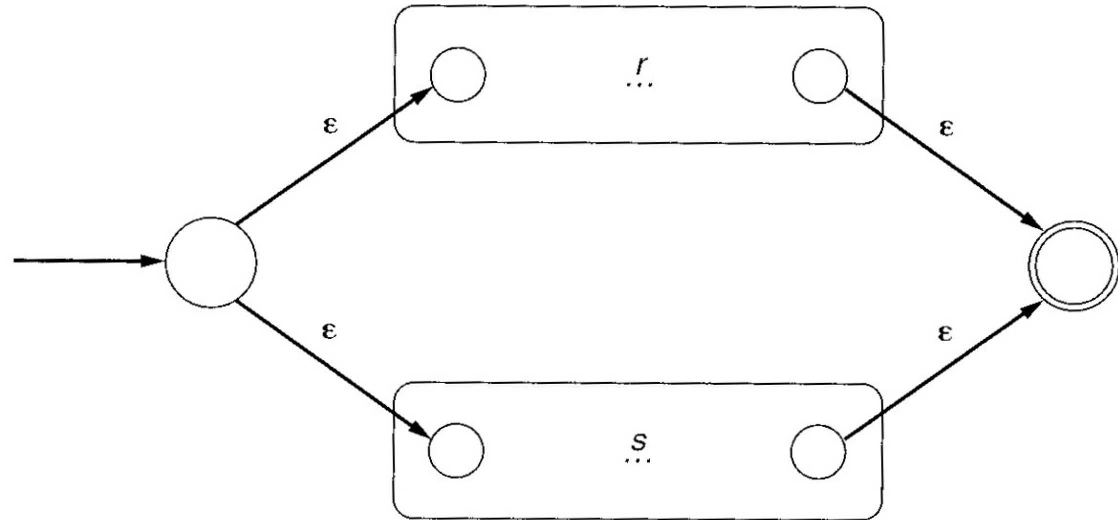
ε:

rs:

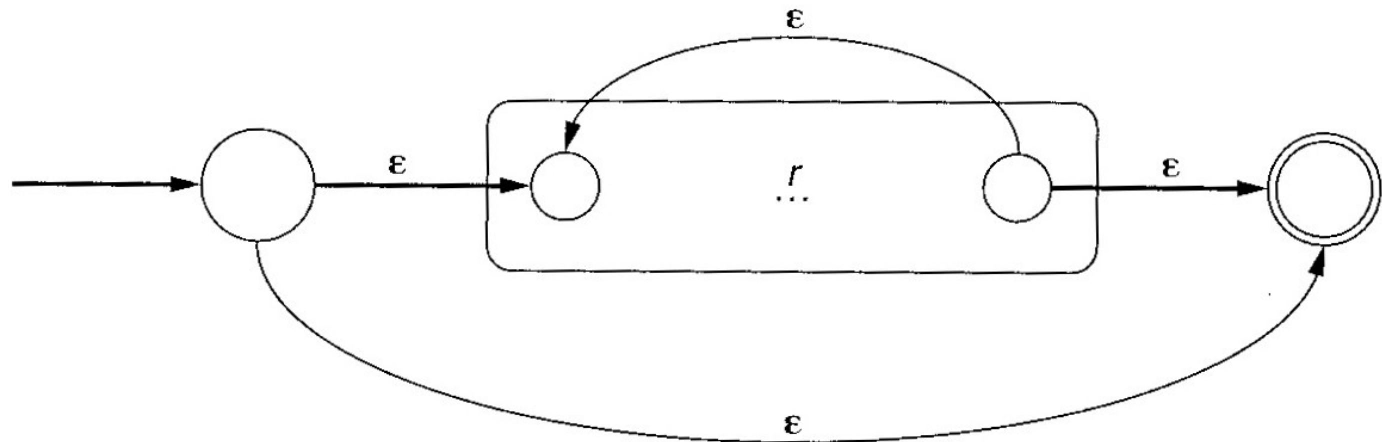


Thomson-konstruksjon II

r | s:



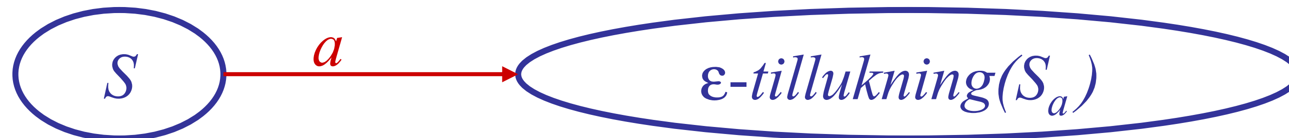
r*:



Transformasjon fra NFA: M til DFA: M'

ALGORITME

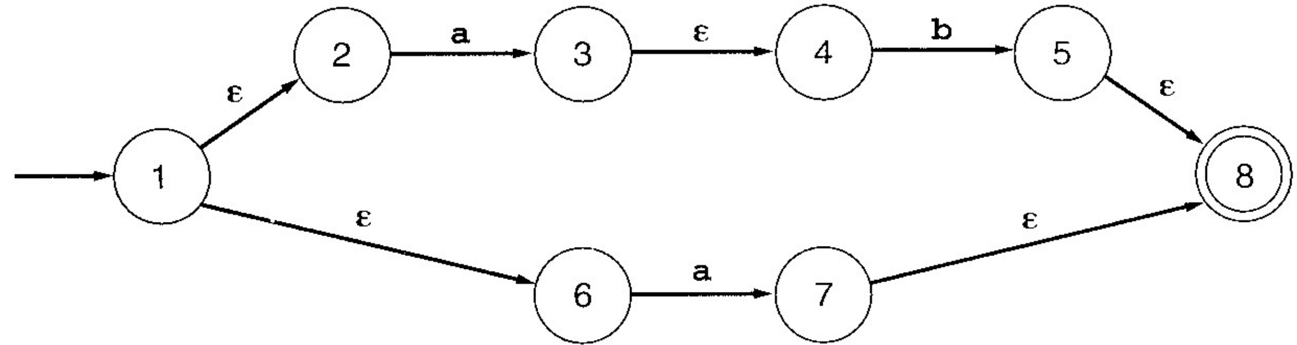
1. Lag start-tilstanden i M' som ε-tillukningen av {start-tilstanden i M}
2. Om S er en tilstand i M', så gjelder:

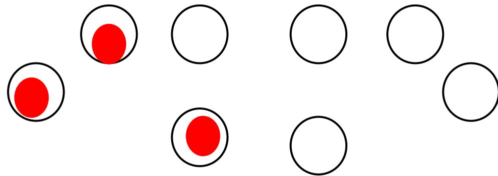


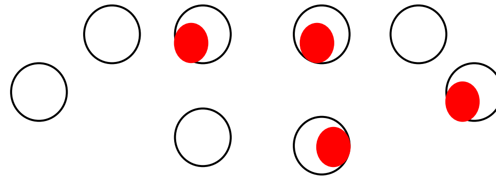
(Må eventuelt opprette ny M'-tilstand)

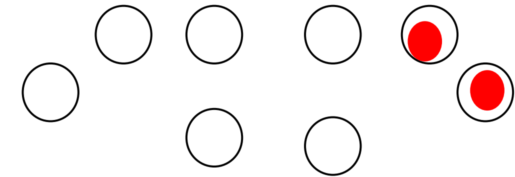
3. Gjenta steg 2 for alle S i M' og alle a i alfabetet til M' ikke får flere nye tilstander.
4. De aksepterende tilstander i M' er de S i M' som inneholder minst én av de aksepterende tilstander i M.

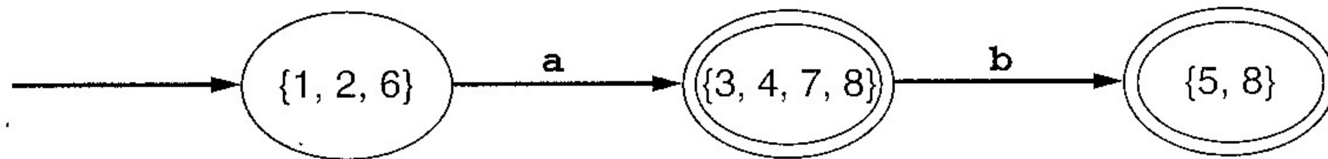
Eksempel 2.16 ab | a





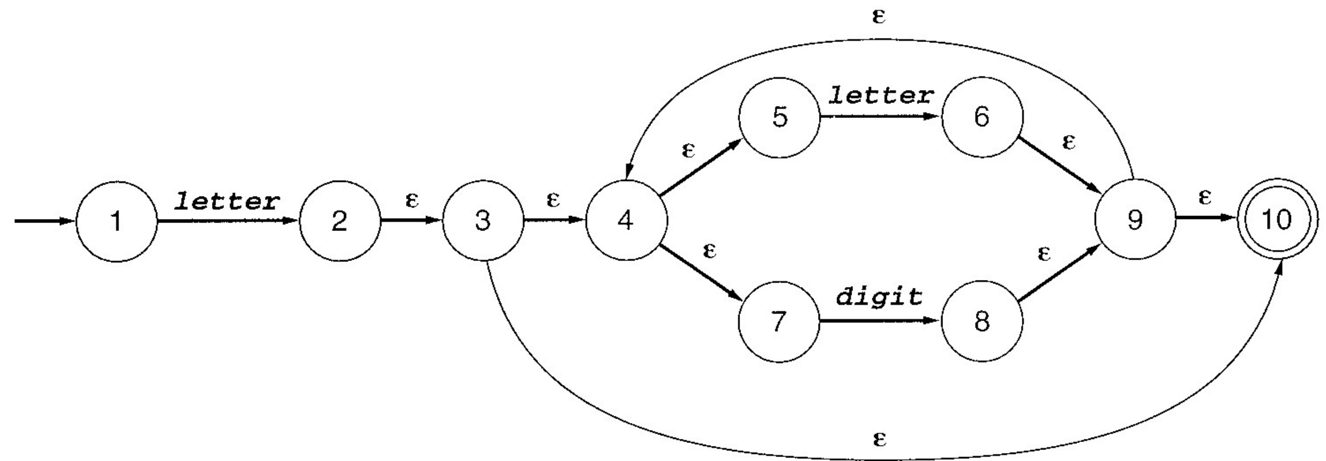


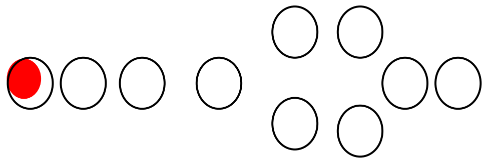


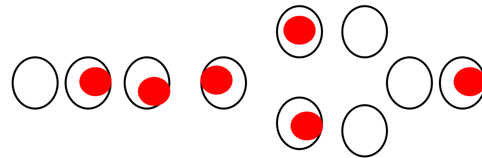


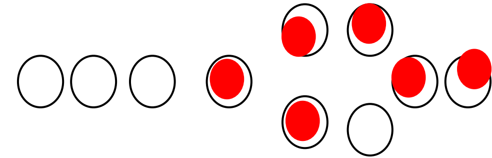
Eksempel 2.17

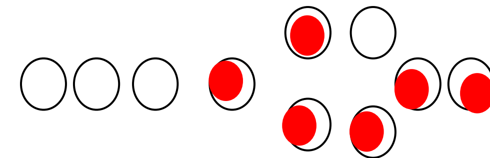
letter (letter | digit)*

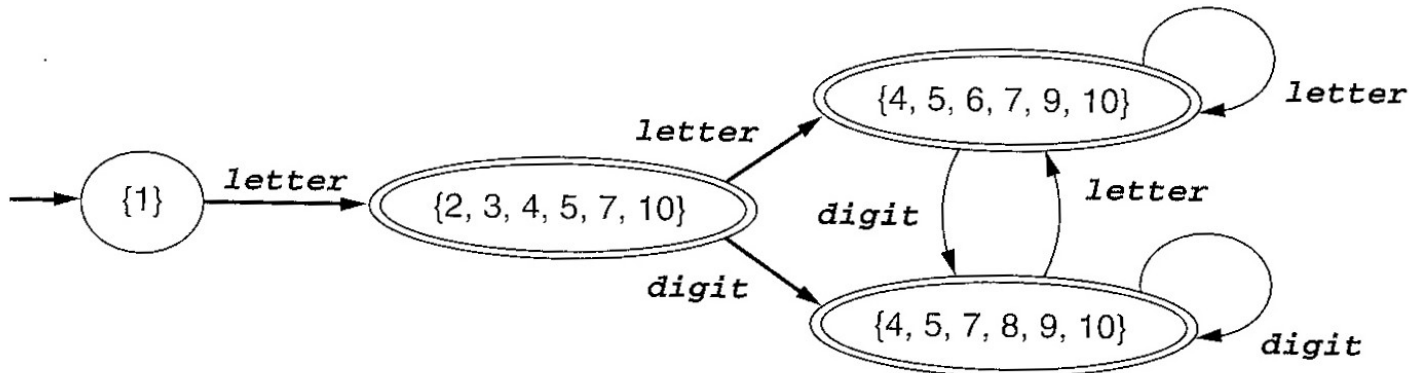










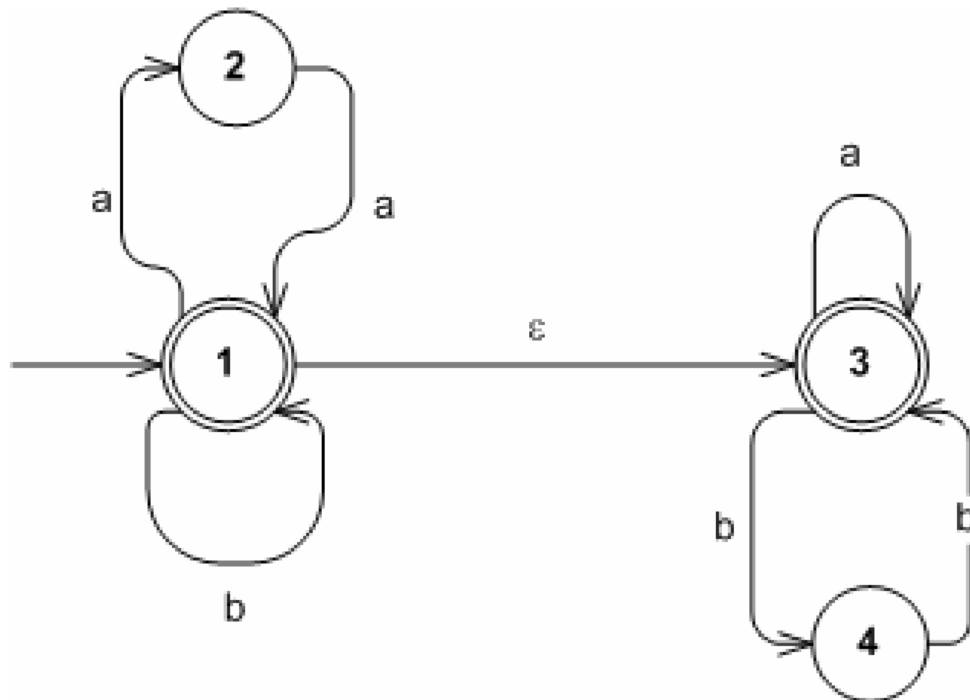


INF 5110 - 2006

Eksamen 2005

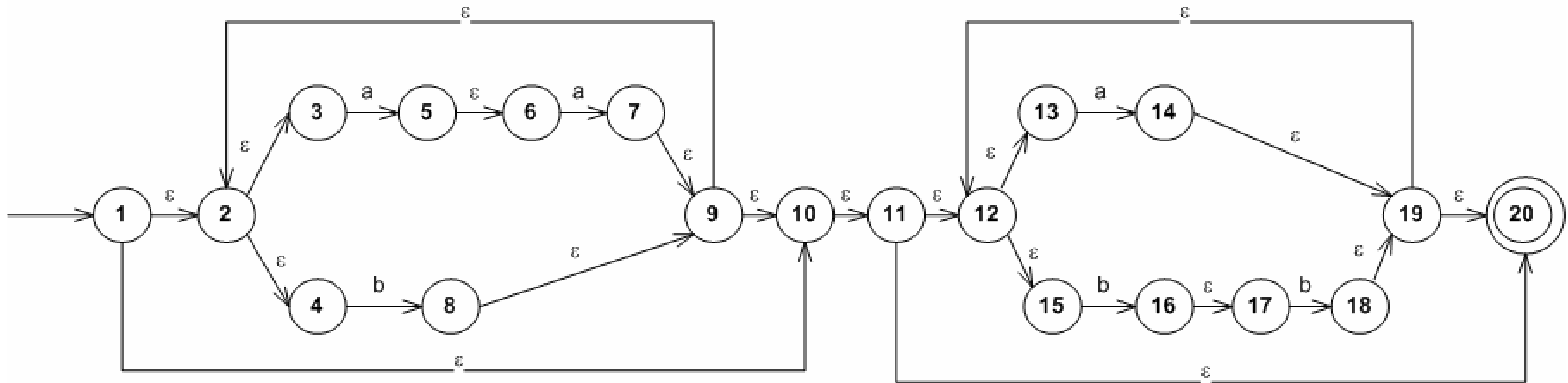
a) Bruk Thompson's konstruksjon til å lage en NFA for det regulære uttrykket **(aa|b)*(a|cc)***b)

b) Skriv følgende NFA ut som et regulært uttrykk:



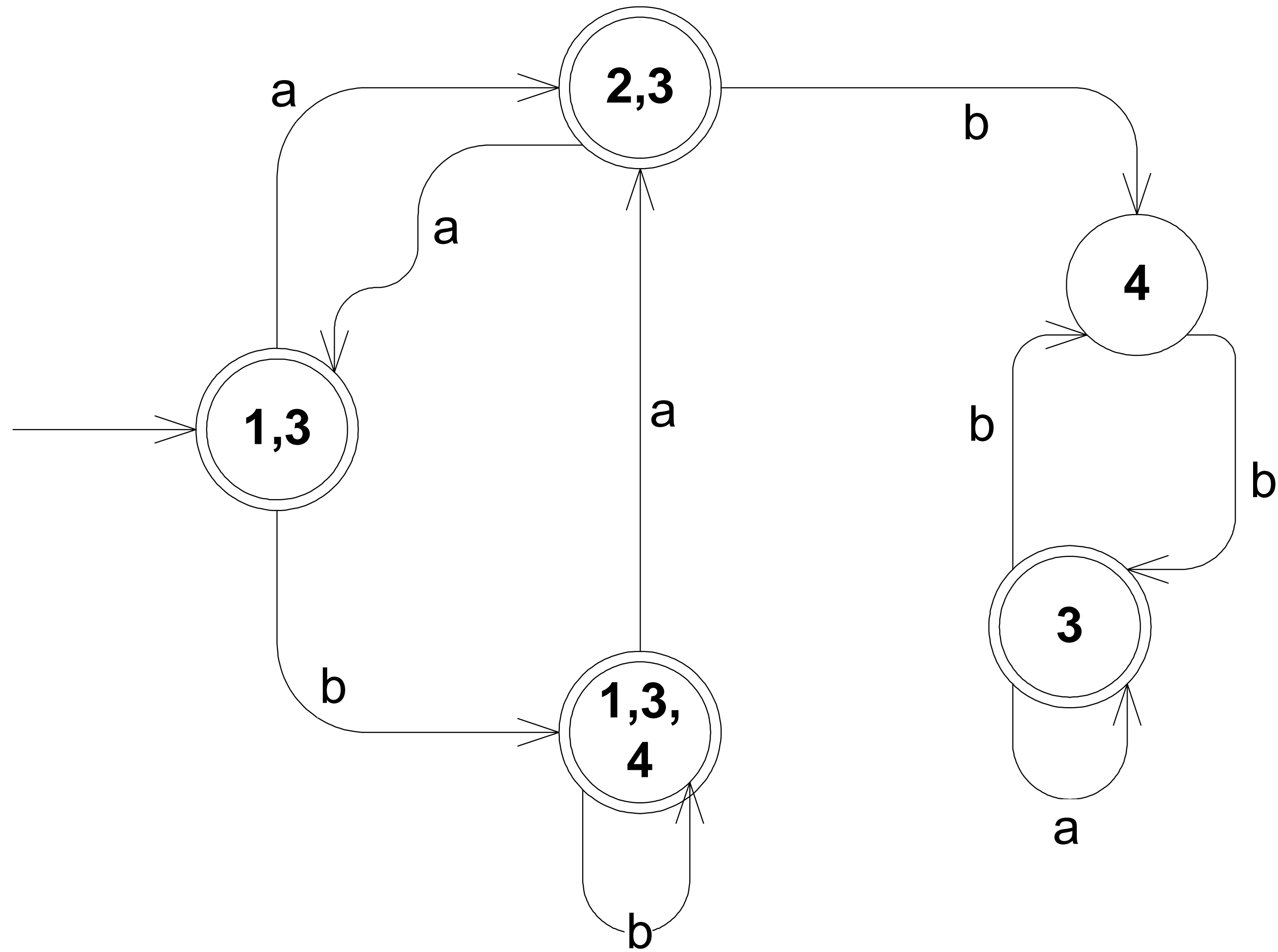
c) Gjør om NFAen fra pkt b) til en DFA.

Thomson-konstruksjon



(aa|b)*(a|bb)*

Fra NFA til DFA



INF 5110 - 2006

Attributt-grammatikker

$exp \rightarrow exp$ **+** $term \mid exp$ **-** $term \mid term$
$term \rightarrow term$ ***** $factor \mid factor$
$factor \rightarrow$ **(** exp **)** $\mid$ **number**

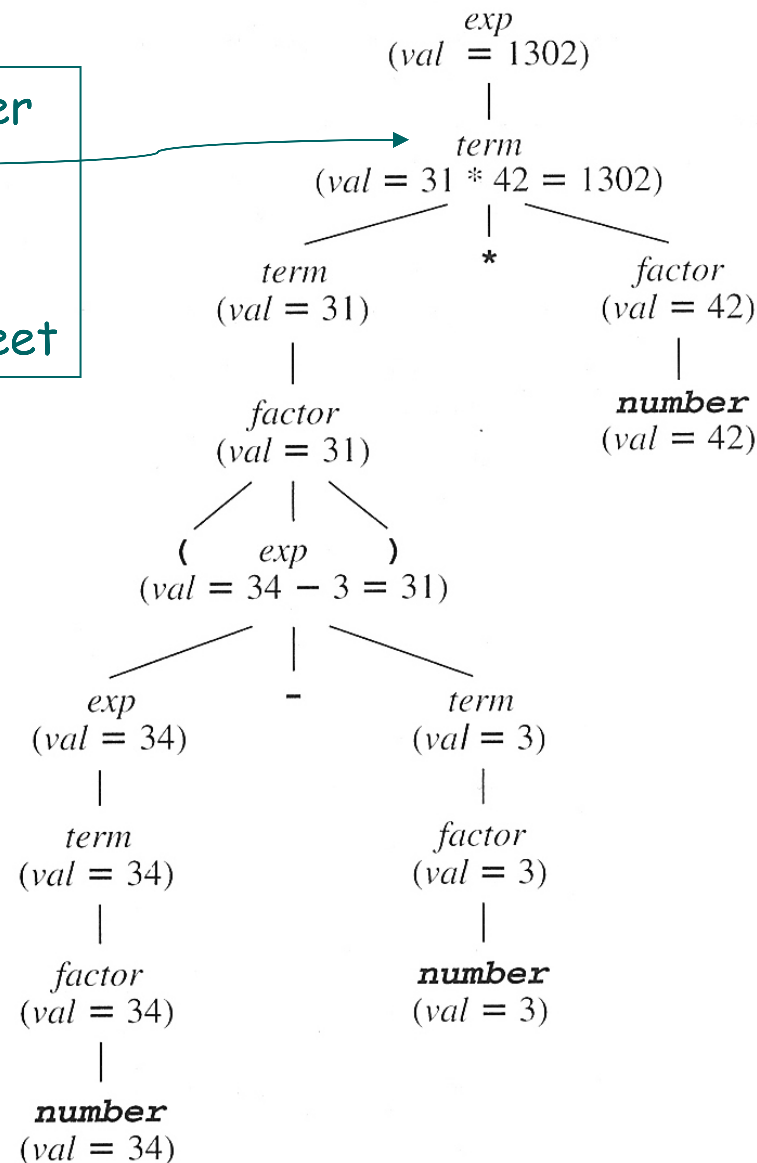
Attributter er variable knyttet til nodene i parseringstreet

Hver semantisk regel er knyttet til en produksjon

Deres verdier definert ved semantiske regler

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ **+** $term$	$exp_1 .val = exp_2 .val + term.val$
$exp_1 \rightarrow exp_2$ **-** $term$	$exp_1 .val = exp_2 .val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2$ ***** $factor$	$term_1 .val = term_2 .val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow$ **(** exp **)**	$factor.val = exp.val$
$factor \rightarrow$ **number**	$factor.val =$ **number**$.val$

Arvede/syntetiserte attributter

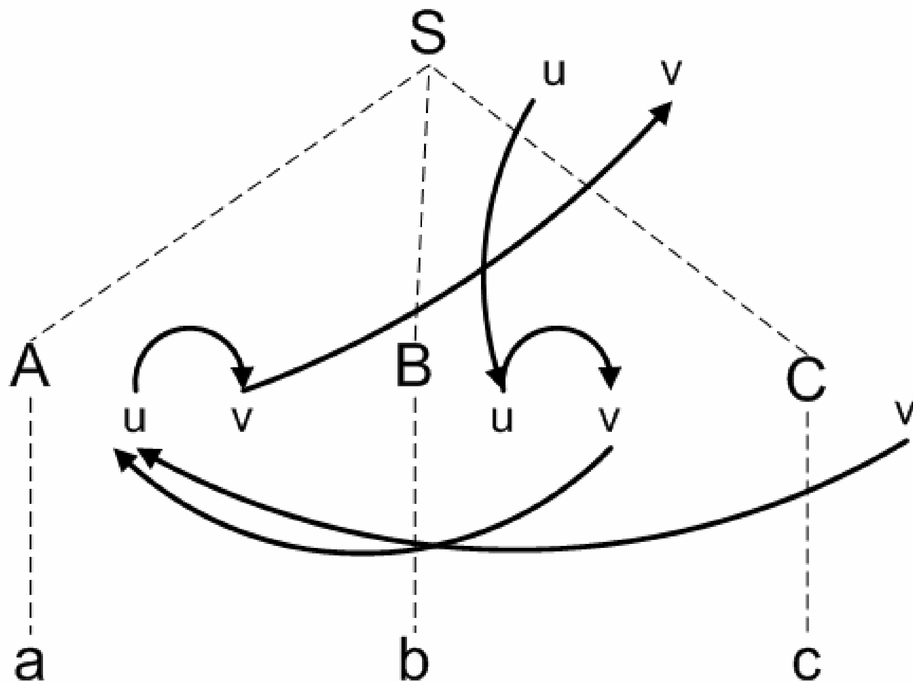


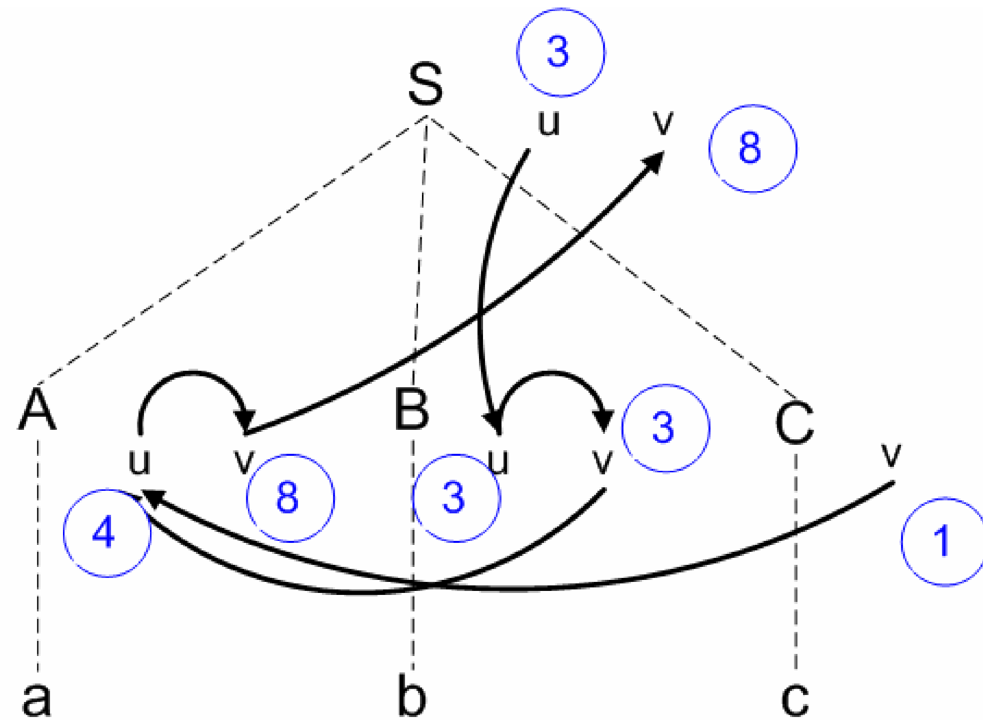
6.13 Consider the following attribute grammar:

Grammar Rule	Semantic Rules
$S \rightarrow A\ B\ C$	$B.u = S.u$
	$A.u = B.v + C.v$
	$S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = 1$

a. Draw the parse tree for the string *abc* (the only string in the language), and draw the dependency graph for the associated attributes. Describe a correct order for the evaluation of the attributes.

b. Suppose that $S.u$ is assigned the value 3 before attribute evaluation begins. What is the value of $S.v$ when evaluation has finished?





Datatyper og typesjekking

- Om typer generelt
  - Hva er typer?
  - Statisk og dynamisk typing
  - Hvordan beskrive typer syntaktisk?
  - Hvordan lagre dem i kompilatoren?
- Gjennomgang av noen typer
  - Grunntyper
  - Type-konstruktører
  - Verdisett og operasjoner
  - Lagring under utførelse (mer i kap 7)
  - Run-time tester og spesielle problemer: array/record/union/peker/...
- Hva vil det si at to variableuttrykk har samme type?
- Hvordan utføre selve type-sjekkingen?

Sjekking av type-riktighet for uttrykk, program, etc

$program \rightarrow var\text{-}decls$ **;** $stmts$

$var\text{-}decls \rightarrow var\text{-}decls$ **;** $var\text{-}decl \mid var\text{-}decl$

$var\text{-}decl \rightarrow$ **id** **:** $type\text{-}exp$

$type\text{-}exp \rightarrow$ **int** $\mid$ **bool** $\mid$ **array [num] of** $type\text{-}exp$

$stmts \rightarrow stmts$ **;** $stmt \mid stmt$

$stmt \rightarrow$ **if** exp **then** $stmt \mid$ **id** **:=** exp

$exp \rightarrow exp + exp \mid exp$ **or** $exp \mid exp$ [exp]

INF 5110 - 2006

Sjekking av type vha attributtgrammatikk

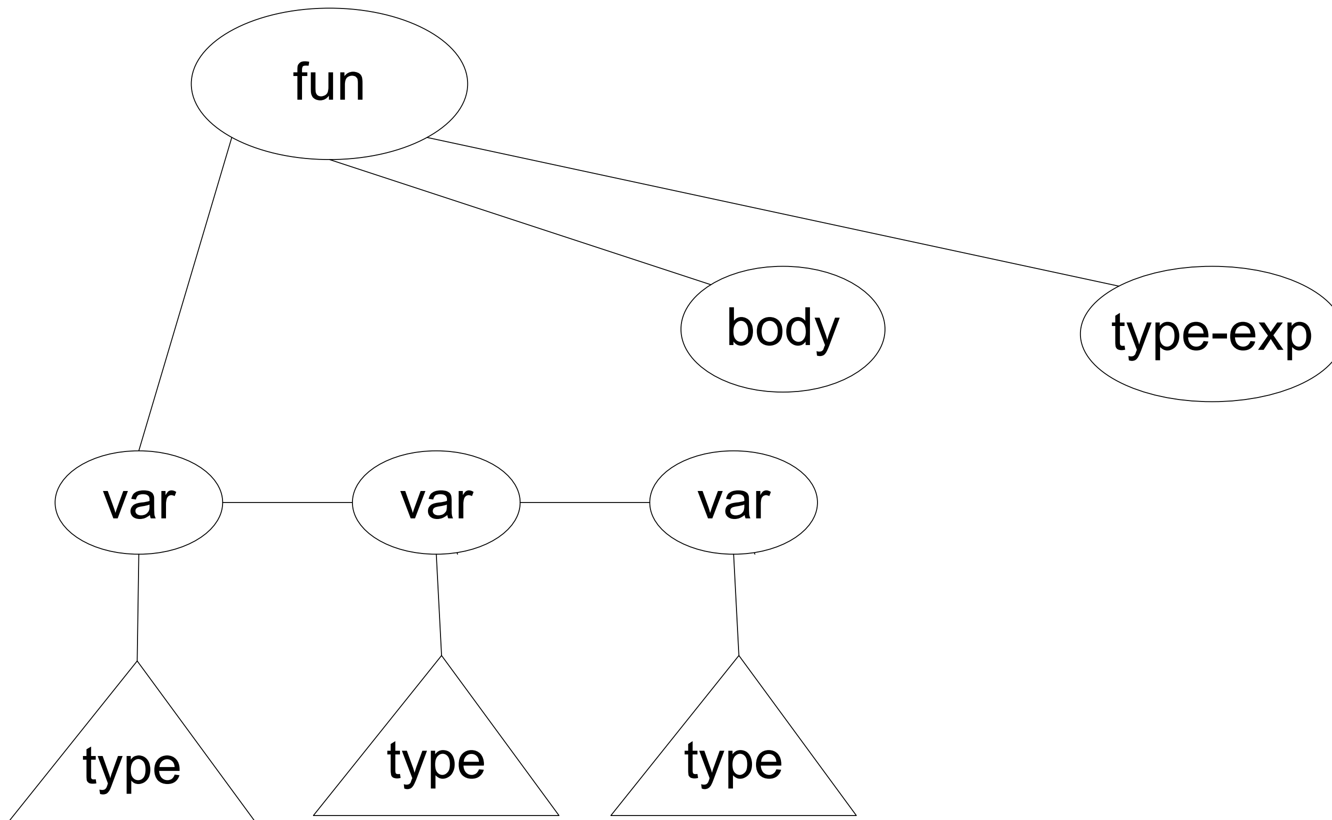
Grammar Rule	Semantic Rules
var-decl → **id** : *type-exp*	*insert*(**id** .*name*, *type-exp.type*)
type-exp → **int**	*type-exp.type* := *integer*
type-exp → **bool**	*type-exp.type* := *boolean*
$type\text{-}exp_1$ → **array** [**num**] **of** $type\text{-}exp_2$	$type\text{-}exp_1$.*type* := *makeTypeNode*(*array*, **num** .*size*, $type\text{-}exp_2$.*type*)
stmt → **if** *exp* **then** *stmt*	**if not** *typeEqual*(*exp.type*, *boolean*) **then** *type-error*(*stmt*)
stmt → **id** := *exp*	**if not** *typeEqual*(*lookup*(**id** .*name*), *exp.type*) **then** *type-error*(*stmt*)
exp_1 → exp_2 + exp_3	**if not** (*typeEqual*(exp_2 .*type*, *integer*) **and** *typeEqual*(exp_3 .*type*, *integer*)) **then** *type-error*(exp_1) ; exp_1 .*type* := *integer*
exp_1 → exp_2 **or** exp_3	**if not** (*typeEqual*(exp_2 .*type*, *boolean*) **and** *typeEqual*(exp_3 .*type*, *boolean*)) **then** *type-error*(exp_1) ; exp_1 .*type* := *boolean*
exp_1 → exp_2 [exp_3]	**if** *isArrayType*(exp_2 .*type*) **and** *typeEqual*(exp_3 .*type*, *integer*) **then** exp_1 .*type* := exp_2 .*type.child1* **else** *type-error*(exp_1)
exp → **num**	*exp.type* := *integer*
exp → **true**	*exp.type* := *boolean*
exp → **false**	*exp.type* := *boolean*
exp → **id**	*exp.type* := *lookup*(**id** .*name*)

6.21

program → *var-decls* **;** *fun-decls* **;** *stmts*
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id :** *type-exp*
type-exp → **int** | **bool** | **array [num] of** *type-exp*
fun-decls → **fun id (** *var-decls* **) :** *type-exp* **;** *body*
body → *exp*
stmts → *stmts* **;** *stmt* | *stmt*
stmt → **if** *exp* **then** *stmt* | **id :=** *exp*
exp → *exp* **+** *exp* | *exp* **or** *exp* | *exp* **[** *exp* **]** | **id (** *exps* **)**
| **num** | **true** | **false** | **id**
exps → *exps* **,** *exp* | *exp*

a. Devise a suitable tree structure for the new function type structures, and write a *typeEqual* function for two function types.

b. Write semantic rules for the type checking of function declarations and function calls (represented by the rule *exp* → **id (** *exps* **)**), similar to the rules of Table 6.10 (page 330).


fun
body
type-exp
var
var
var
type
type
type

Grammar Rule	Semantic Rule
fun-decls → **fun id (** *var-decls* **):** *type-exp* **;** *body*	*fun-decls.type* = *makeTypeNode (fun, var-decls.types, type-exp.type)*
exp → **id (** *exps* **)**	**if** *isFunType(lookup(**id**.name))* **and** *exps.types = parameterTypesOf(**id**.name)* **then** *exp.type = lookup(**id**.name)* **else** *type-error*

- Forutsetter at
  - *var-decls.types* defineres som en liste av de typer, som de enkelte var-decl bidrar med;
  - *exps.types* defineres som listen av typene til de enkelte exp i listen av exp;
  - funksjonen *parameterTypesOf* gir tilsvarende listen av de typer som finnes i TypeNode for funksjonen

Eksamensoppgave 2004

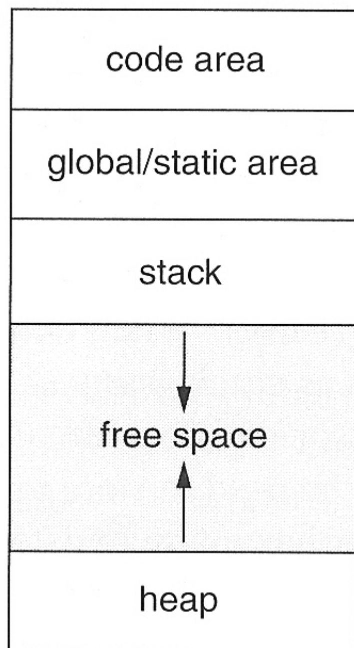
class → **class** ***name*** *superclass* **{** *decls* **}**
decls → *decls* **;** *decl* | *decl*
decl → *variable-decl*
decl → *method-decl*
method-decl → *type* ***name*** **(** *params* **)** *body*
type → **int** |**bool** | **void**
superclass → ***name***

Metoder med samme navn som klassen er ’konstruktører’,
og det gjelder følgende regel:
Konstruktører må være spesifisert med typen **void.**

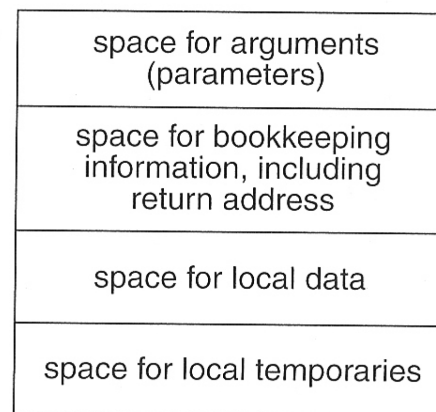
Grammar Rule	Semantic Rule
class → **class name** { *decls* }	*decls.enclosingClassName* = **name**.name
$decls_1$ → $decls_2$; *decl*	$decls_2$.*enclosingClassName* = $decls_1$.*enclosingClassName* *decl.enclosingClassName* = $decls_1$.*enclosingClassName*
decls → *decl*	*decl.enclosingClassName* = *decls.enclosingClassName*
decl → *variable-decl*	
decl → *method-decl*	*method-decl.enclosingClassName* = *decl.enclosingClassName*
method-decl → *type* **name** (*params*) *body*	***if*** (**name**.name = *method-decl.enclosingClassName*) ***then if*** (**not**(*type*.type = void))***then*** *error*("*constructor not of type void*") eller ***if*** (**name**.name = *method-decl.enclosingClassName*) ***and*** (**not**(*type*.type = void))***then*** *error*("*constructor not of type void*")
type → **int**	*type*.type = int
type → **bool**	*type*.type = bool
type → **void**	*type*.type = void

Lagerorganisering

- Typisk organisering under utførelse dersom et programmeringsspråk har alle slags data (statisk, stakk, dynamisk)



- Typisk organisering av data for et prosedyrekall (aktiveringsblokk)



Det er gjerne ut fra plasseringen her man karakteriserer språk til være
- statisk organisert
- stakk-organisert
- heap/dynamisk organisert

INF 5110 - 2006

Aktiveringsstakk

```
#include <stdio.h>

int x,y;

int gcd( int u, int v)
{ if (v == 0) return u;
  else return gcd(v,u % v);
}

main()
{ scanf("%d%d",&x,&y);
  printf("%d\n",gcd(x,y));
  return 0;
}
```

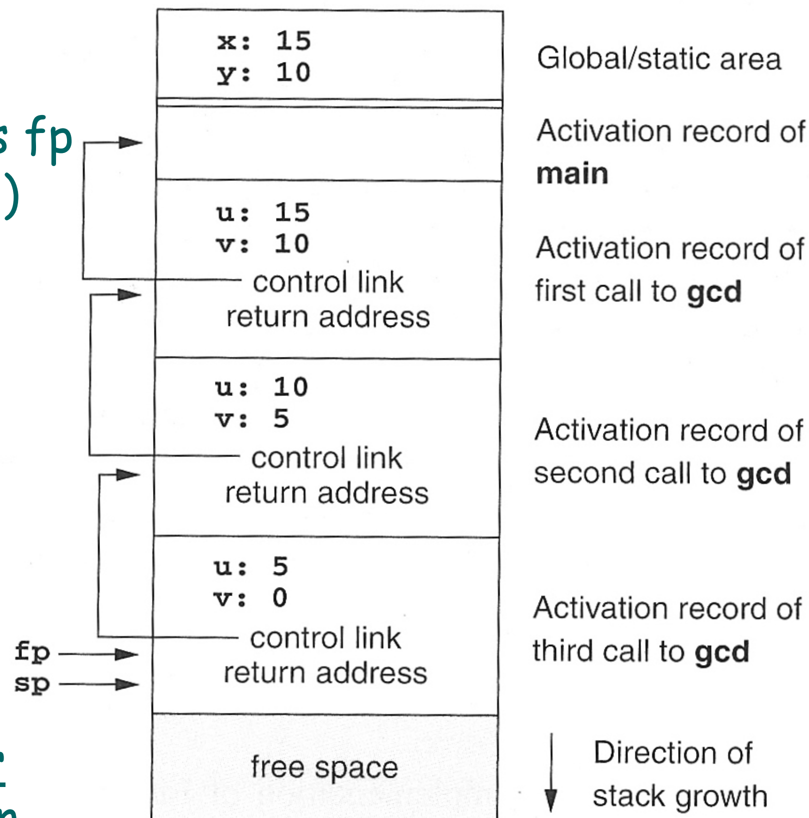
- Aktiverings-blokkene kan organiseres som en stakk. Kreves om man tillater rekursive kall

Return address
Program-adressen man er kalt fra

control link
Angir kallerens fp ('dynamisk link')

frame pointer
Peker på fast sted i den aktuelle aktiveringsblokkken

stack pointer
Angir grensen mellom brukt og ledig lagerareal

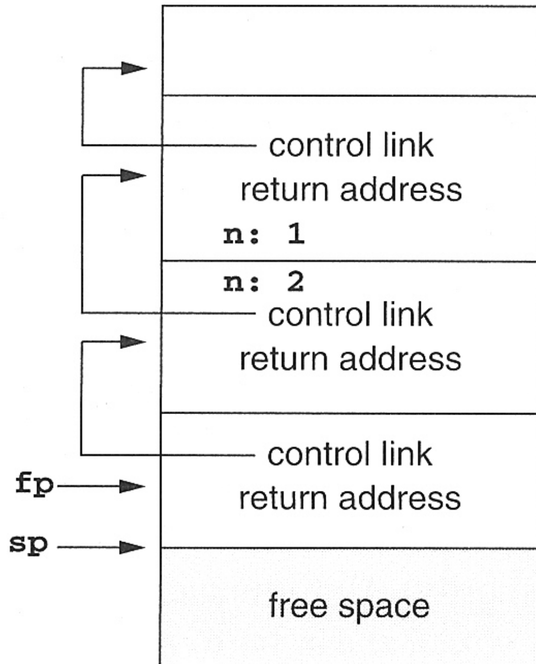


Prosedyrer inne i prosedyrer

- Nestede prosedyrer
- Nested klasser (inner classes) kan behandles på samme måte.

```
program nonLocalRef;

procedure p;
var n: integer;

      procedure q;
      begin
        (* a reference to n is now
          non-local non-global *)
      end; (* q *)

      procedure r(n: integer);
      begin
        q;
      end; (* r *)

begin (* p *)
  n := 1;
  r(2);
end; (* p *)

begin (* main *)
  p;
end.
```

Et første forsøk



Activation record of main program

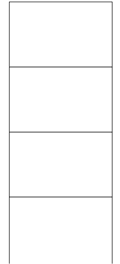
Activation record of call to **p**

Activation record of call to **r**

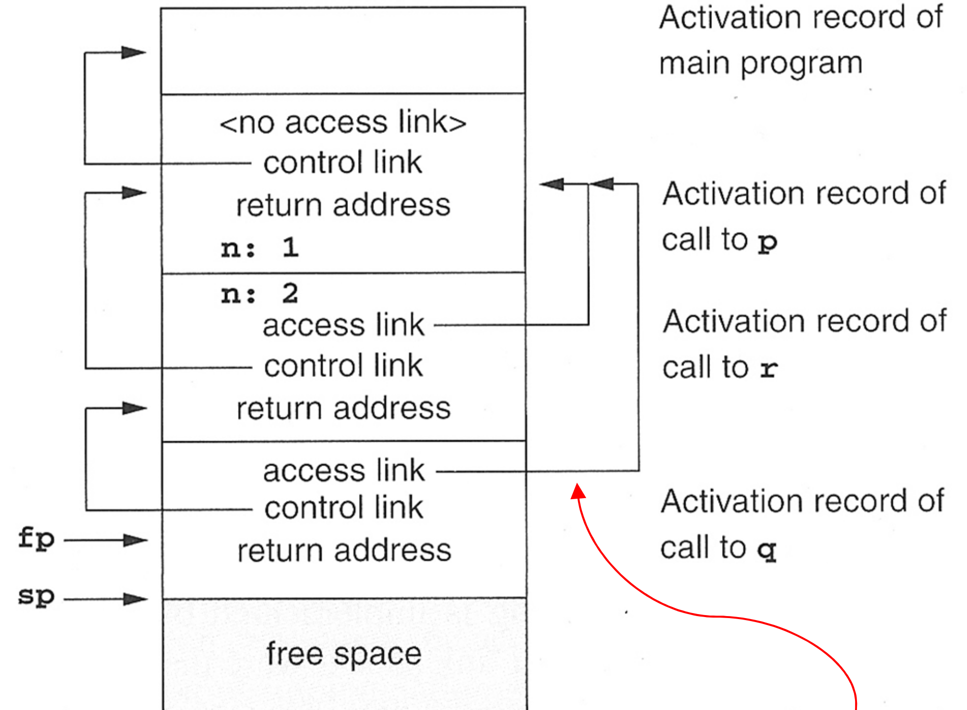
Activation record of call to **q**

Hvordan kan vi aksessere 'n' i 'p' ?

Kontekstvektor ('display')



Vi trenger noe ekstra (aksess-link/statisk link)

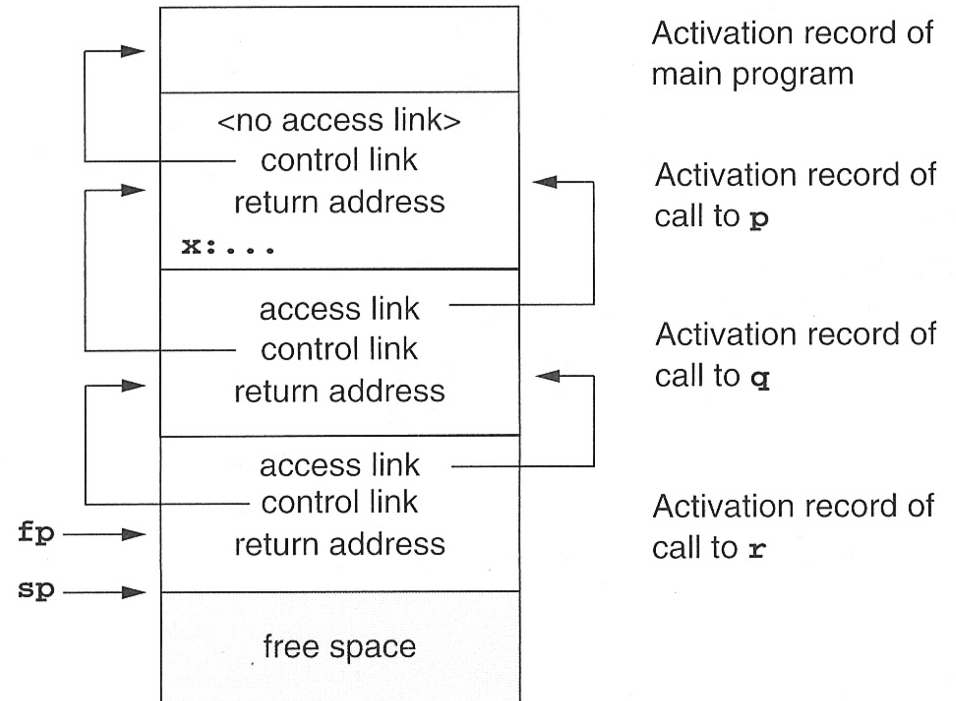


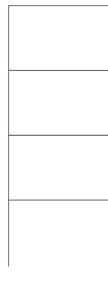
Går altid til aktuell utgave av tekstlig omgivelse

Eksempel med flere nivåer

```
program chain;

procedure p;
var x: integer;

   procedure q;
      procedure r;
      begin
         x := 2;
         ...
         if ... then p;
      end; (* r *)
   begin
     r;
   end; (* q *)

begin
  q;
end; (* p *)

begin (* main *)
  p;
end.
```

Program-blokkene får da et blokk-nivå





fp.al.al.x

diff i blokknivå

INF 5110 - 2006

Oppgave 3.b (2004)

Spørsmål 3.b

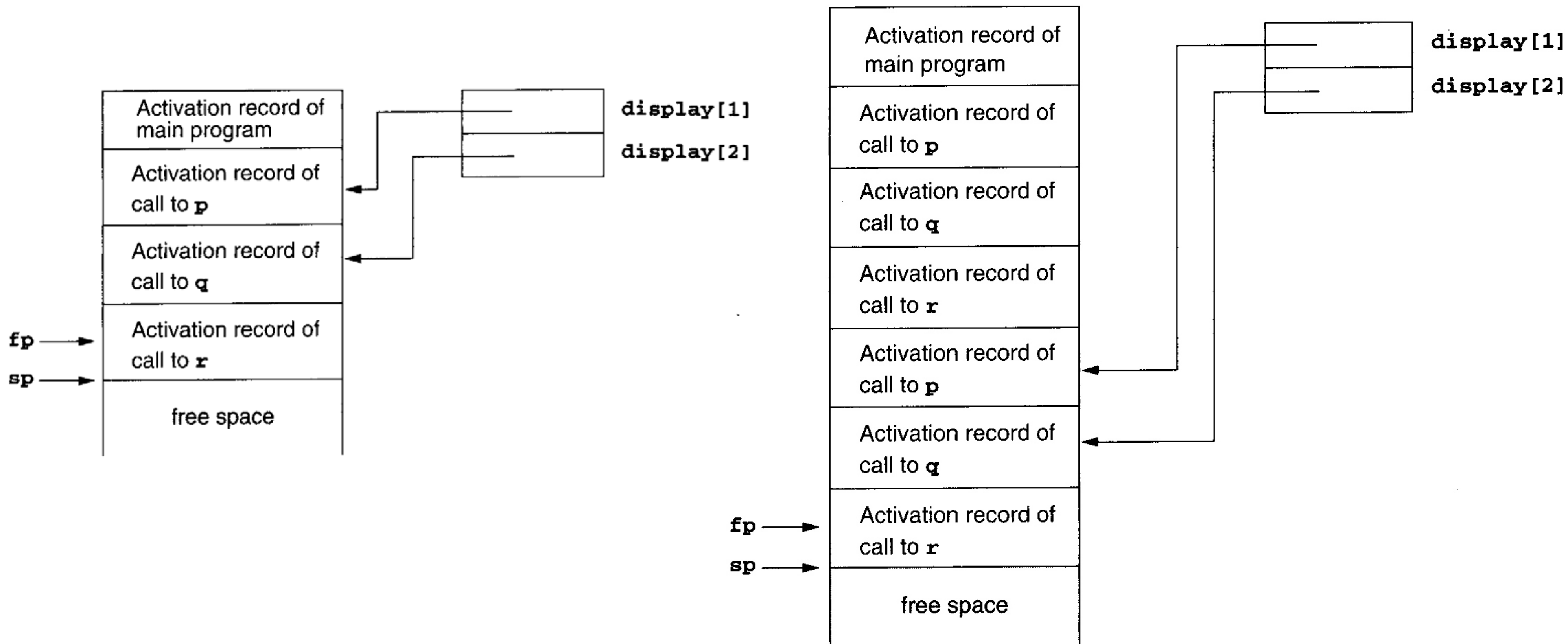
Vi antar et språk som skissert i spørsmål 3.a, og at vi bruker en standard implementasjon. Setningen ``x:= 2;'' står lokalt i en prosedyre P. P er deklarert i en prosedyre Q, og Q er deklarert i en prosedyre R. Variablen *x* er deklarert lokalt i R.

Angi hva setningen over (og spesielt aksessen til *x*) blir oversatt til. Du kan bruke ``dot-notasjon'', med en liten forklaring til.

```
void R() {
  int x;
  ...;
  void Q() {
      ...;
      void P() {
         ...;
         x := 2;
         ...
      }
  }
}
```

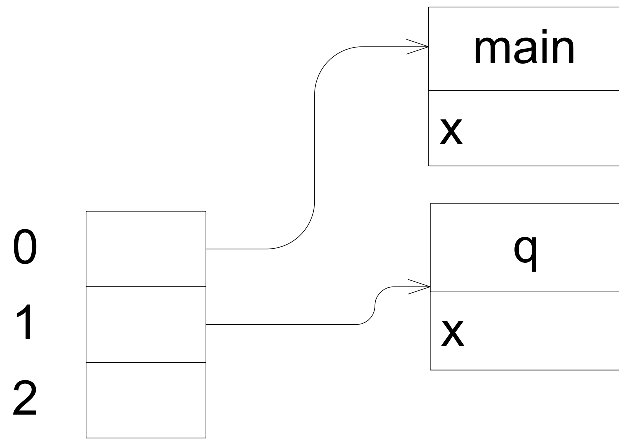
fp.al.al.x

Oppg 7.10d

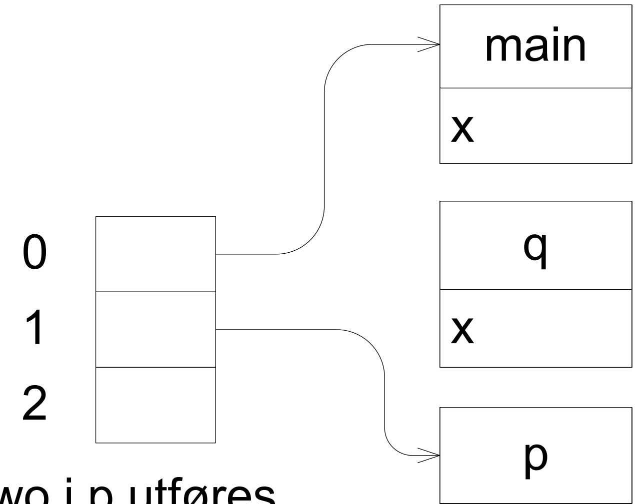


d. A problem exists in using a display in a language with procedure parameters. Describe the problem using Exercise 7.5.

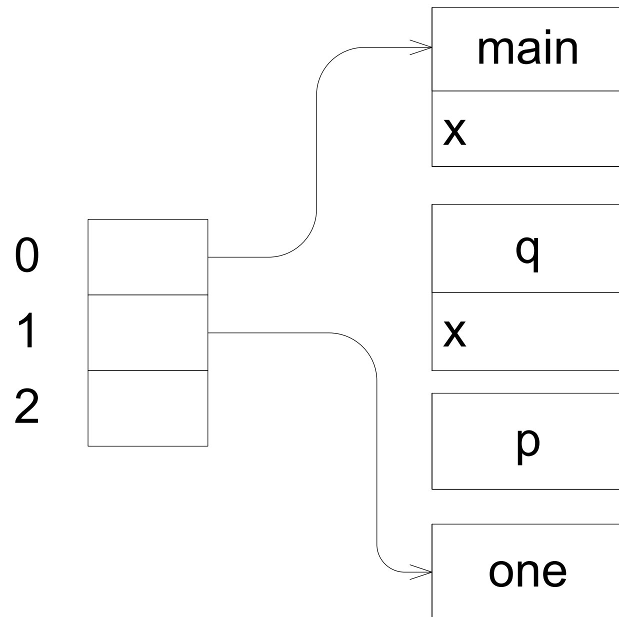
Mens q i main utføres



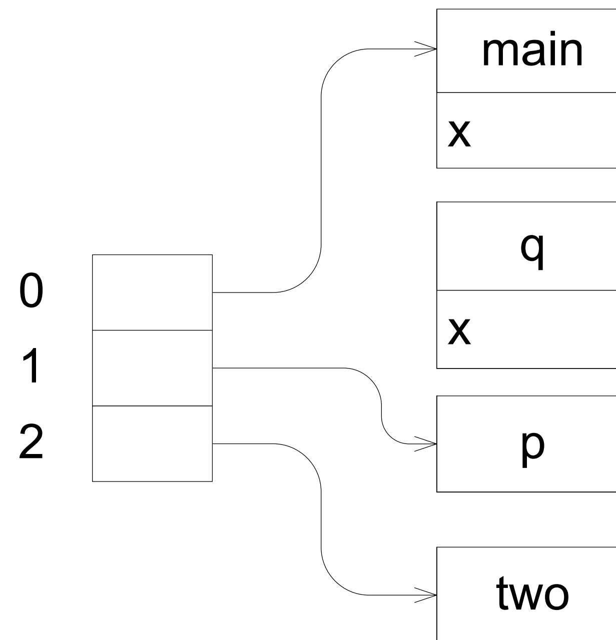
Mens p i q utføres



Mens one i p utføres

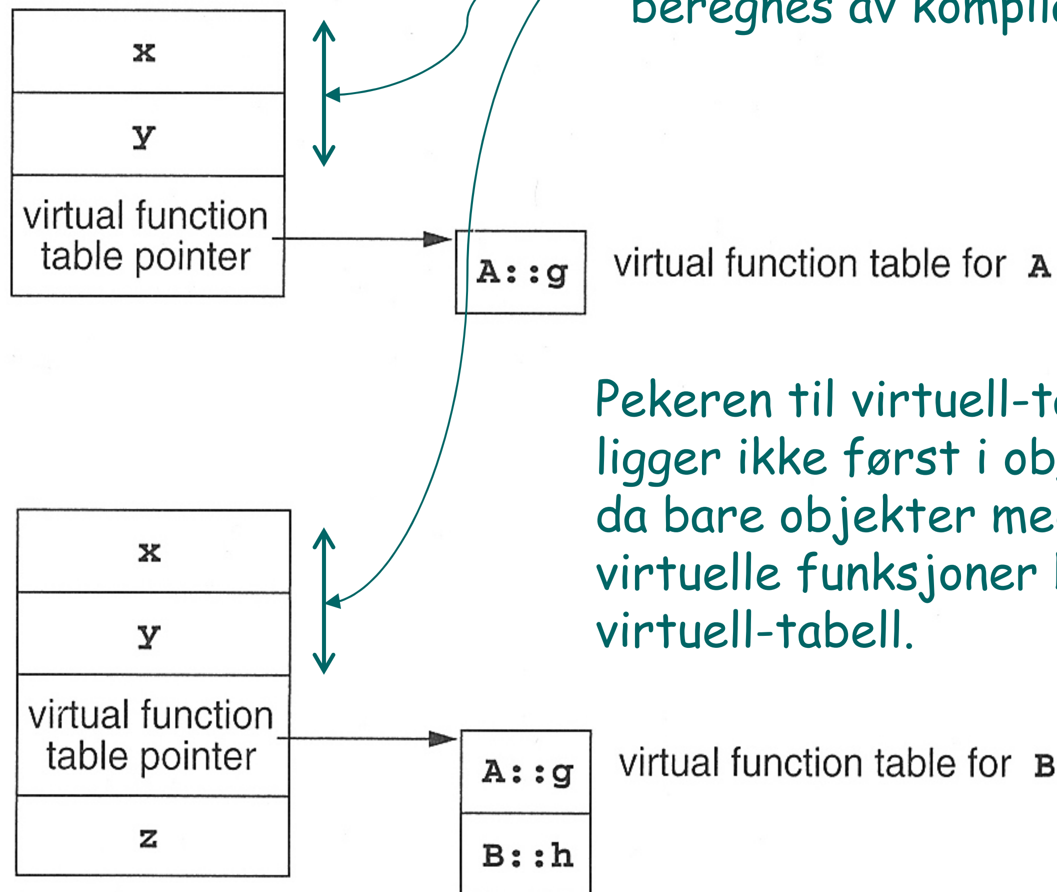


Mens two i p utføres



Impl. av virtuelle metoder

```
class A
{ public:
  double x,y;
  void f();
  virtual void g();
};

class B: public A
{ public:
  double z;
  void f();
  virtual void h();
};
```



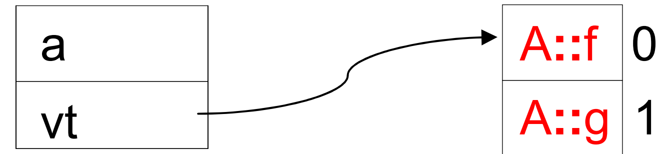
Virtuell-tabell pekeren har relativ-adresse som kan beregnes av kompilatoren

Pekeren til virtuell-tabellen ligger ikke først i objektet, da bare objekter med virtuelle funksjoner har en virtuell-tabell.

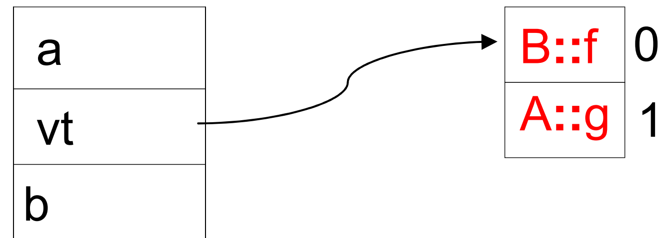
7.13 Draw the memory layout of objects of the following C++ classes, together with the virtual function tables as described in Section 7.4.2:

```
class A
{ public:
  int a;
  virtual void f();
  virtual void g();
};
class B : public A
{ public:
  int b;
  virtual void f();
  void h();
};
class C : public B
{ public:
  int c;
  virtual void g();
}
```

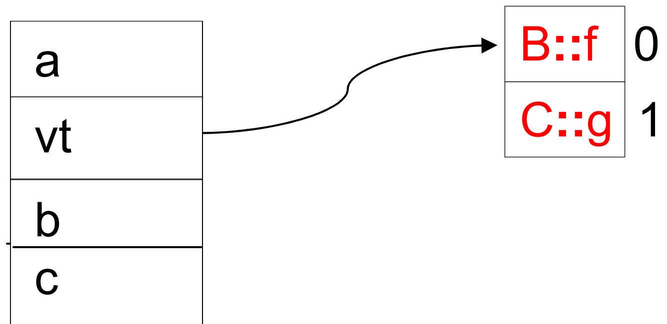
A-objekt



B-objekt



C-objekt

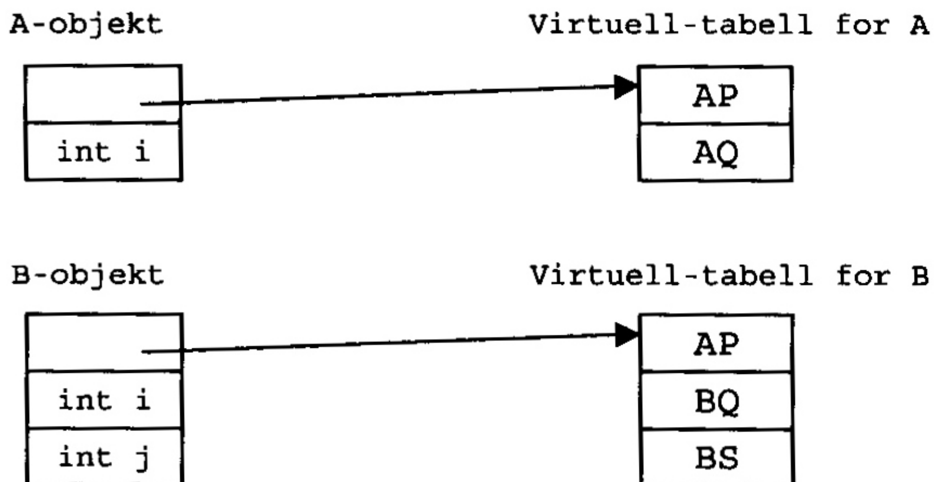


Oppg. 3c (2004)

Spørsmål 3.c

Vi har følgende klasse-deklarasjoner i et språk der alle prosedyrer (metoder, funksjoner, ...) i klasser er virtuelle, og der peker-variable er typet med klasser på standard måte:

```
class A {
  integer i;
  procedure P( ) { ... AP ...}
  procedure Q( ) { ... AQ ...}
}

class B extends A {
  integer j;
  procedure Q( ) { ... BQ ...}
  procedure S( ) { ... BS ...}
}
```



Tegn et A- og et B-objekt, og vis hvordan virtuell-tabellene (``virtual function tables'') til A og B ser ut, og hvor det ligger pekere til disse tabellene. Angi innholdet av virtuell-tabellene.

Parameteroverføring

- 'by value'
  - Hver formell parameter blir implementert som en lokal variabel i prosedyren
    - Ved kallet gjøres formell_var = aktuell_var
- 'by reference'
  - Overfører en peker til den aktuelle variable
- 'by value-result'
  - Det allokeres en lokal varabel, som ved 'by value'
    - Ved kallet gjøres formell_var = aktuell_var
    - Ved retur utføres aktuell_var = formell_var
- 'by name'
  - Den aktuelle parameteren blir substituert inn for den formelle ('nesten' rent tekstlig: den aktuelle parameteren beholder sitt skop, så altså ikke makro-ekspansjon)

7.16

Give the output of the following program (in C syntax) using the four parameter passing methods of Section 7.5:

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}

main()
{ int a[3] = {1,2,0};
   swap(i,a[i]);
   printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
   return 0;
}
```

by value	by reference
0 1 2 0	1 0 2 0
by value-result	**by name**
1 0 2 0 1 1 0 0	2 1 -1 0

INF 5110 - 2006