

# Symboltabellen

- Programmeringsspråk har
  - Deklarasjoner, som definerer navn
    - Konstant-deklarasjon
    - Type-deklarasjon
    - Variabel-deklarasjon
    - Prosedyre-deklarasjon
    - Klasse-deklarasjon
  - Bruksforekomster av navn, f.eks. i uttrykk
    - Skal bindes opp til en deklarasjon av samme navn
- Symboltabellen
  - Holder orden på de deklarasjons-navn som gjelder på det stedet man er i programmet
  - Har funksjonen 'lookup(bruks-navn), som gir den deklarasjon, som navnet skal bindes til (og hvis dette ikke lykkes: udeklart)

# Symboltabellen – II

- To hovedfilosofier

1. Tradisjonell tabell

- lookup(id)
- insert(id)
- delete(id)

Brukes til oppdatere tabellen ved passering av dekl. og ved inngang/utgang av blokker

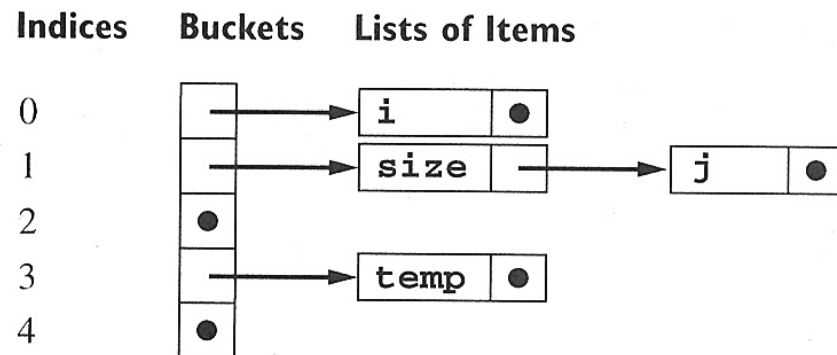
```
{ int i; ... double d;
  void p(...)
  { int i;
    ...
  }
  int j
}
```

2. Selve syntakstreet

- Look-up blir da en lete-prosess
- Insert/delete blir implisitte (alt etter hvordan man flytter seg i treet)
- Kan være vanskelig å få lookup effektiv

# Ved bruk av tradisjonell tabell

- er hashing en grei effektiviserings-metode



- I pakkene: også info om deklarasjonen
- For senere: Nye pakker settes inn først i listen

```
{
  int temp;
  int j;
  real i
  void size(...)
    { ...
    }
}
```

# Eksempler

```
int i,j;

int f(int size)
{ char i, temp;
  ...
  { double j;
    ...
  }
  ...
  { char * j;
    ...
  }
}
```

```
program Ex;
var i,j: integer;

function f(size: integer): integer;
var i,temp: char;

  procedure g;
  var j: real;
  begin
    ...
  end;

  procedure h;
  var j: ^char;
  begin
    ...
  end;

begin (* f *)
  ...
end;

begin (* main program *)
  ...
end.
```

# Varianter av blokkstruktur

- Hva slag konstruksjoner kan ligge inne i hverandre
  - program, klasse, prosedyre, ...
- Hvor dyb kan blokkstrukturen bli?
- Må navne angis (ikke nødvendigvis deklarereres) før de brukes?
- Hva kan aksesseres utenfra ved `<peker>.<egenskap>?`
  
- Algol, Simula
- C, C++
- Pascal
- Java, C#

# 'Hashing

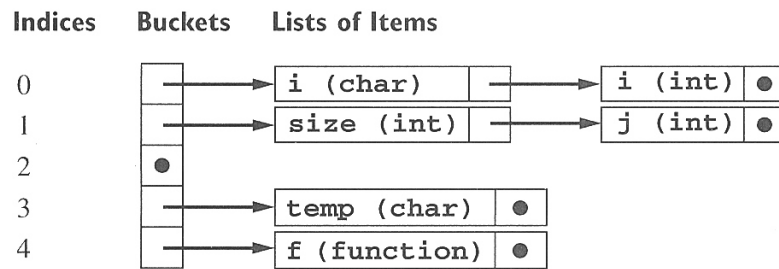
- med stakk for blokkstruktur og tradisjonell tabell

```

int i,j;

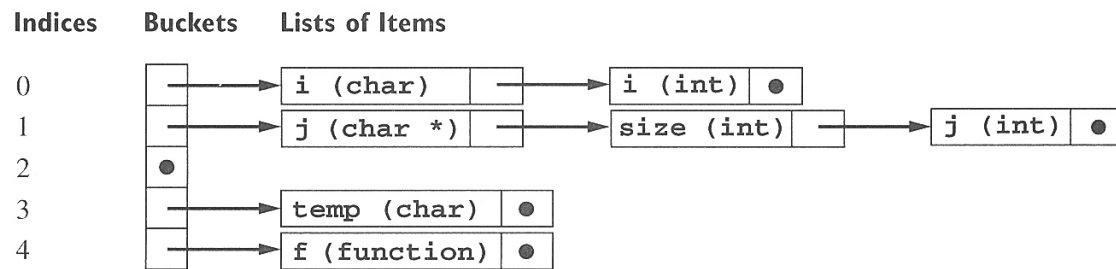
int f(int size)
{ char i, temp;
  ...
  { double j;
    ...
  }
  ...
  { char * j;
    ...
  }
}

```



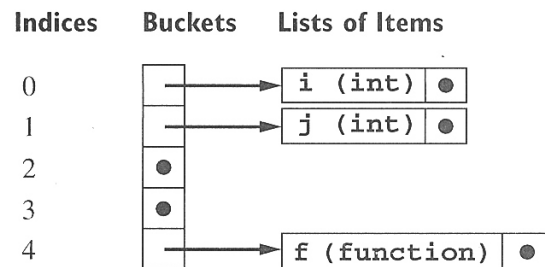
(a) After processing the declarations of the body of f

Hvis navne må angis før bruk: deklarasjoner settes inn etter hvert



(b) After processing the declaration of the second nested compound statement within the body of f

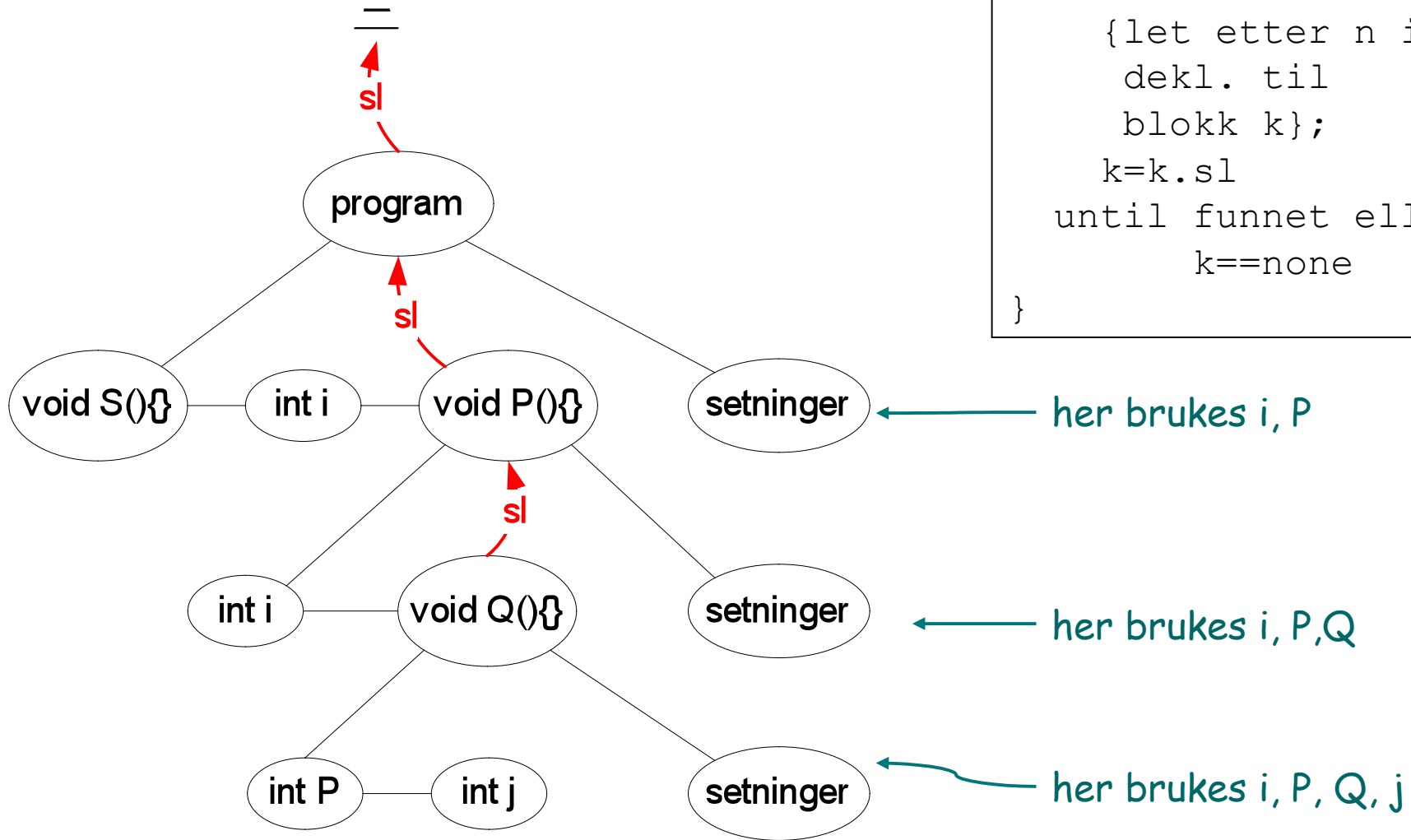
Ellers: to gjennomløp  
Ved blokkstart: Alle blokkens deklarasjoner settes inn



(c) After exiting the body of f (and deleting its declarations)

# Bruk av syntakstre til lookup

```
lookup(n) {  
  k=nåværende blokk  
  do  
    {let etter n i  
     dekl. til  
     blokk k};  
    k=k.sl  
  until funnet eller  
    k==none  
}
```



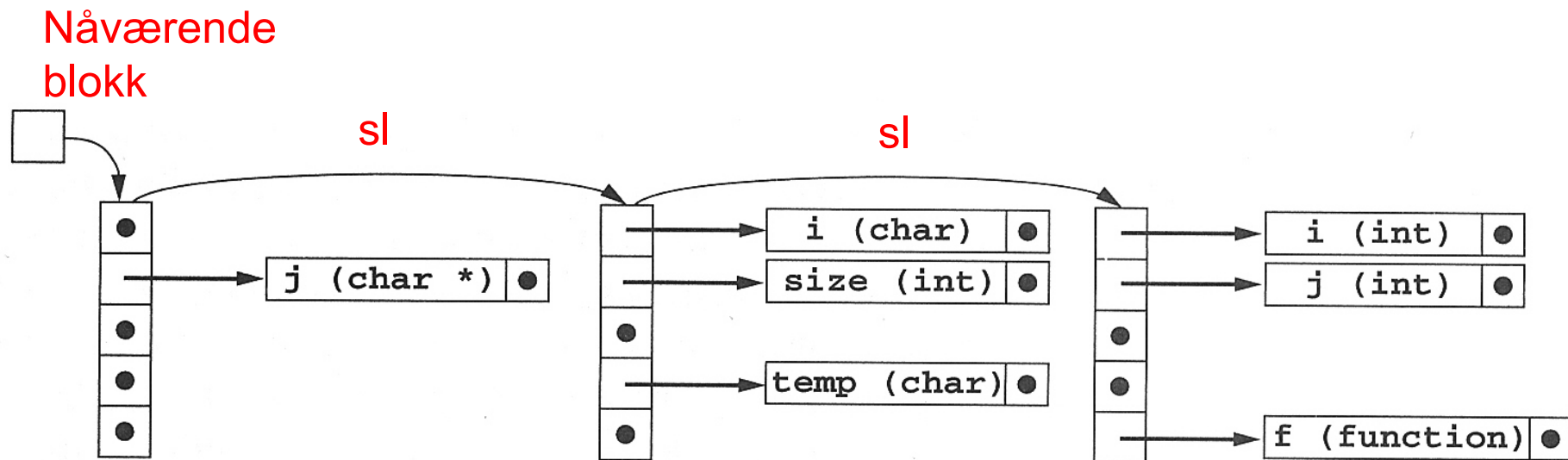
# Ved bruk av syntakstre til lookup

- Ved deklarasjon før bruk
  - Man gjør oppslagene etter hvert som treet bygges opp
  - Ett gjennomløp
- Ellers
  - Bygg ferdig hele treet i ett gjennomløp
  - Gå gjennom treet en gang til og gjør lookup for hver bruksforekomst (ut fra det stedet forekomsten er)



# En mellomløsning

- Samler deklarasjonene i hver blokk i hver sin tabell
- Bruker hashing i hver tabell
- Bruker statisk link pekere, og implementerer lookup ved leting



# Definisjoner utenfor skop

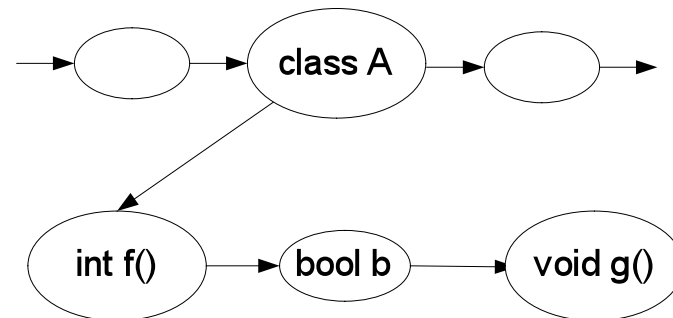
```
class A
{ ... int f();...}; // f is a member function
```

```
A::f() // this is the definition of f in A
{ ... }
```

- For å implementere dette (og lignende ting, som f.eks. remote aksess) må man ha deklarasjonene for hver blokk i egne tabeller

```
A a;
...
a.f();
...
```

```
class A{
  int f({...});
  bool b;
  void g(){...}
}
```



# Scope - I

```
typedef int i;  
int i;
```

```
int gcd(int n, int m)  
{ if (m == 0) return n;  
  else return gcd(m, n % m);  
}
```

```
int i = 1;  
  
void f(void)  
{ int i = 2, j = i+1;  
  ...  
}  
...
```

```
void f(void)  
{... g() ...}  
  
void g(void)  
{... f() ...}
```

Sequential <> Collateral

## Scope - II

```
void g(void); /* function prototype  
                declaration */
```

```
void f(void)  
{... g() ...}
```

```
void g(void)  
{... f() ...}
```

# Dynamisk skop: binde navn via dynamisk link

- Bokens eksempel

```
#include <stdio.h>

int i = 1;

void f(void)
{ printf(“%d\n”, i); }

void main(void)
{ int i = 2;
  f();
  return 0;
}
```

```
void Y() {
  int i;
  void p() {
    int i;
    ...
    Q();
    ...
  }
  void Q() {
    ...
    i = 5;
    ...
  }
  ...
  P();
  ...
}
```

hvilken 'i'  
i  
'i = 5'?

# Navnebinding etc.

- Skrevet ut som attributtgrammatikk
- Brukt til å definere statisk semantikk, ikke til implementasjon

$S \rightarrow exp$

$exp \rightarrow ( exp ) \mid exp + exp \mid id \mid num \mid let\ dec-list\ in\ exp$

$dec-list \rightarrow dec-list , decl \mid decl$

$decl \rightarrow id = exp$

exp: symtab      arvet  
          nestlevel      arvet  
          err              syntetisert

declist: intab      arvet

decl: outtab      syntetisert  
          nestlevel      arvet

```
let x = 2, y = 3 in
  (let x = x+1, y=(let z=3 in x+y+z)
   in (x+y)
  )
```

insert(tab, name, l)  
 isIn(tab, name)  
 lookup(tab, name)

leverer ny tabell  
 ja/nei  
 gir nivået

Grammar Rule	Semantic Rules
$S \rightarrow exp$	$exp.syntab = emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$
$exp_1 \rightarrow exp_2 + exp_3$	$exp_2.syntab = exp_1.syntab$ $exp_3.syntab = exp_1.syntab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$ <b>or</b> $exp_3.err$
$exp_1 \rightarrow ( exp_2 )$	$exp_2.syntab = exp_1.syntab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$
$exp \rightarrow id$	$exp.err = \mathbf{not\ isin}(exp.syntab, id.name)$ } 2
$exp \rightarrow num$	$exp.err = \mathbf{false}$
$exp_1 \rightarrow \mathbf{let\ dec-list\ in\ } exp_2$	$dec-list.intab = exp_1.syntab$ $dec-list.nestlevel = exp_1.nestlevel + 1$ $exp_2.syntab = dec-list.outtab$ } 3 $exp_2.nestlevel = dec-list.nestlevel$ $exp_1.err = (dec-list.outtab = errtab) \mathbf{or\ } exp_2.err$

---

$dec-list_1 \rightarrow dec-list_2 , decl$

$dec-list_2.intab = dec-list_1.intab$   
 $dec-list_2.nestlevel = dec-list_1.nestlevel$   
 $decl.intab = dec-list_2.outtab$   
 $decl.nestlevel = dec-list_2.nestlevel$   
 $dec-list_1.outtab = decl.outtab$

} 4

---

$dec-list \rightarrow decl$

$decl.intab = dec-list.intab$   
 $decl.nestlevel = dec-list.nestlevel$   
 $dec-list.outtab = decl.outtab$

} 4

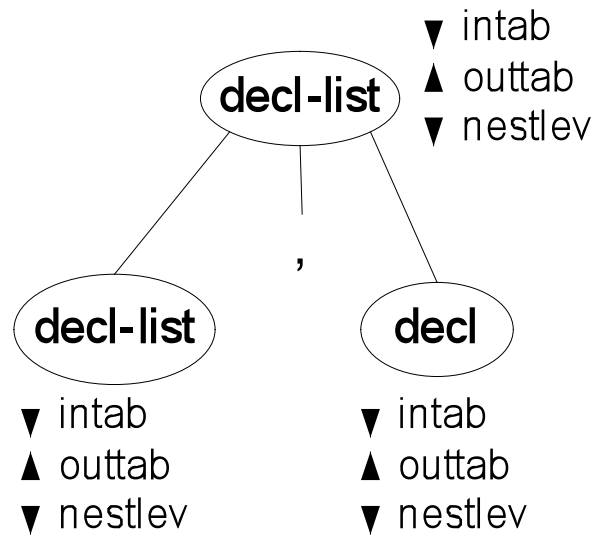
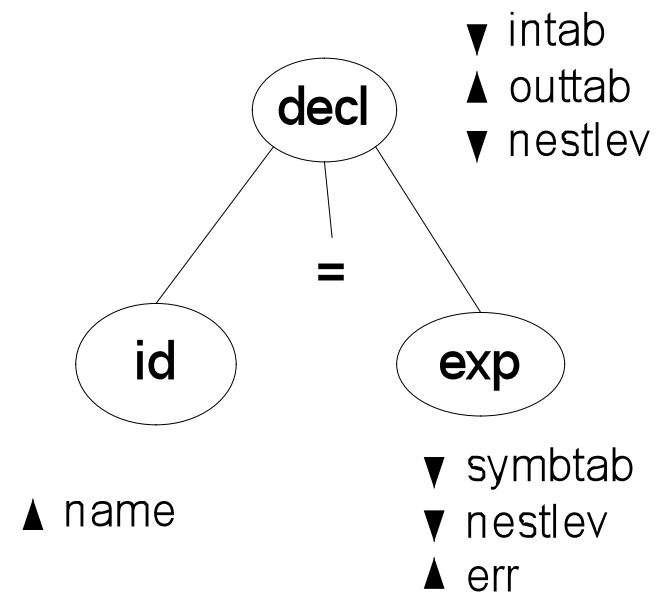
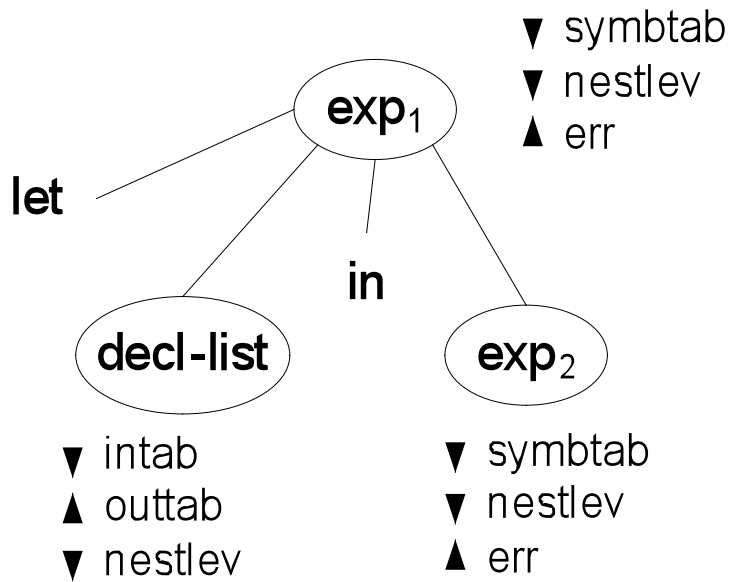
---

$decl \rightarrow \mathbf{id} = exp$

$exp.syntab = decl.intab$   
 $exp.nestlevel = decl.nestlevel$   
 $decl.outtab =$   
    **if** ( $decl.intab = errtab$ ) **or**  $exp.err$   
    **then**  $errtab$   
    **else if** ( $lookup(decl.intab, \mathbf{id}.name) =$   
         $decl.nestlevel$ )  
    **then**  $errtab$   
    **else**  $insert(decl.intab, \mathbf{id}.name, decl.nestlevel)$

} 1





# Noen siste punkter omkring symboltabellen

- Implisitte deklarasjoner (Fortran)
- Navnebinding i kompilatoren gir ikke endelig dynamisk binding
- Kan man ha samme navn på f.eks. variabel og type?
- Overloading
  - Får lov å bruke samme navn på flere ting (også i samme blokk)
  - Må da kunne skille på noe annet, gjerne antall/type av parametre

```
i + j          integer +  
r + s          real +  
  
void f(int i)  
void f(int i, int j)  
void f(double r)
```

# Regler

1. Ikke samme navn på to i samme let-blokk

```
let x=2,x=3 in x+1
```

2. Navne må deklarereres

```
let x=2 in x+y
```

3. 'Innermost' binding

```
let x=2 in (let x=3 in x)
```

4. 'sequential' deklarasjon

```
let x=2,y=x+1 in (let x=x+y,y=x+y in y)
```