INF5110 – 11. og 23. april, 2014
Kap. 8 – kodegenerering

Foilene inneholder også noe ekstra-stoff som ikke står i boka, men som er pensum

Stein Krogdahl,
Ifi UiO

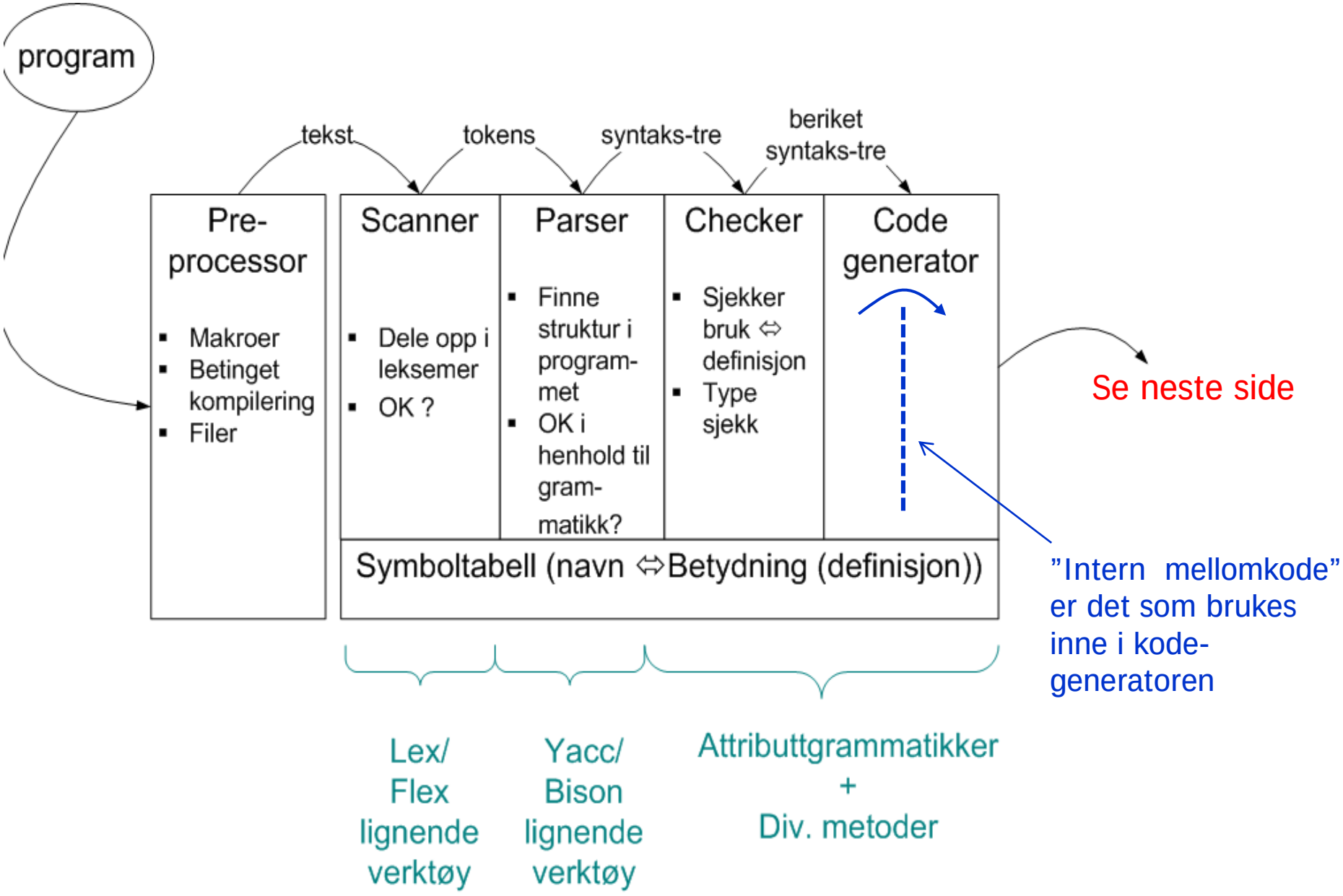
Stoffet fra kap. 8 er nok til Oblig 2

8.1 Bruk av mellomkode
8.2 Basale teknikker for kode- generering
8.3 Kode for aksess av datastrukturer (ikke alt)
8.4 Kode for generereing for kontroll-setninger og logiske uttrykk
Kanskje også noe fra: 8.5 og 8.9

Det blir også noe mer om kodegenerering:

- Registerallokering
- Noe optimalisering

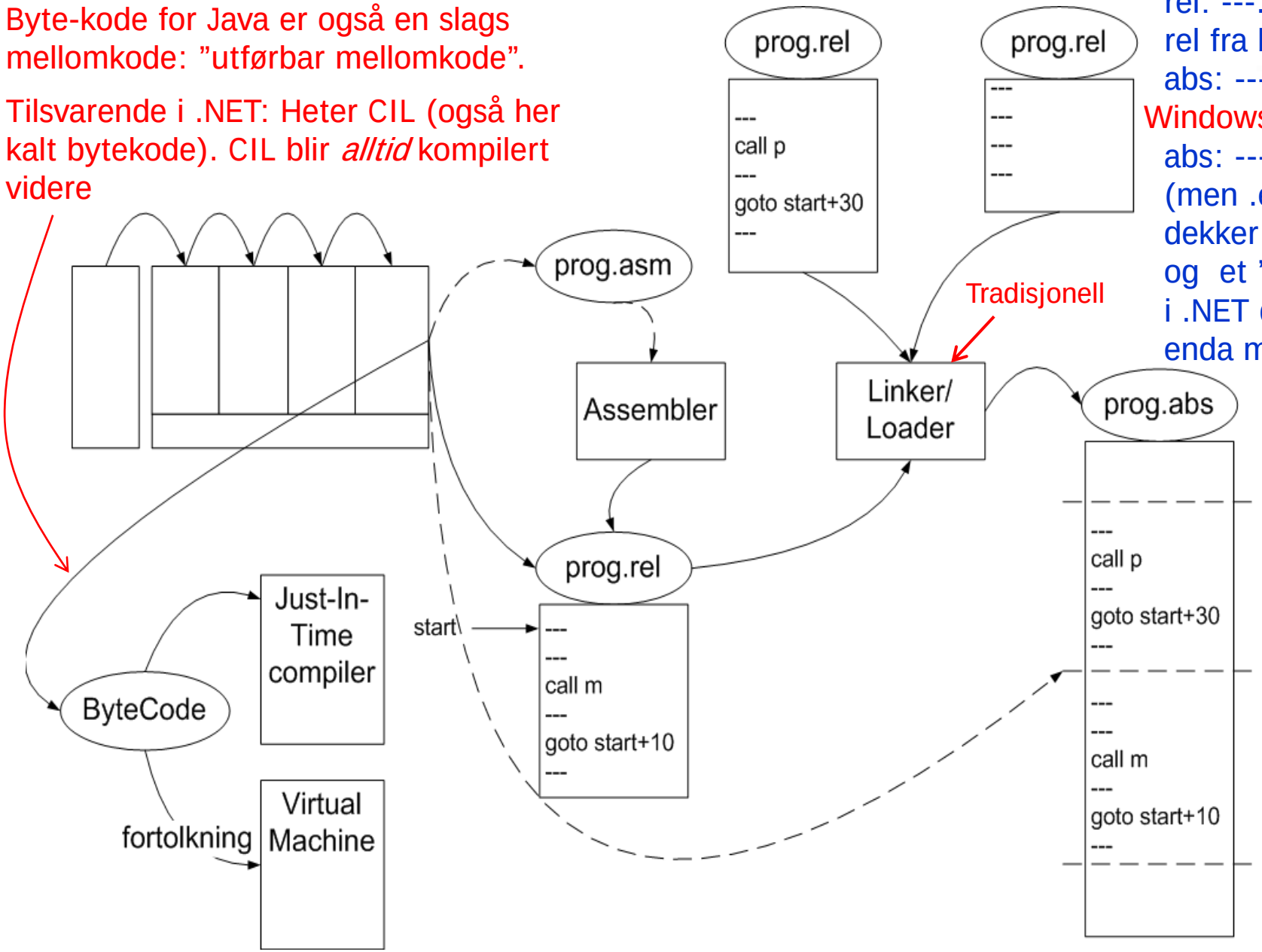
Fra første forelesning: Anatomien til en kompilator



Anatomien til en kompilator - II

Byte-kode for Java er også en slags mellomkode: ”utførbar mellomkode”.

Tilsvarende i .NET: Heter CIL (også her kalt bytekode). CIL blir *alltid* kompilert videre



Unix/Linux:
- asm: ---.s
- rel: ---.o
- rel fra bibl: ---.a
- abs: --- (tomt)

Windows:
- abs: ---.exe
- (men .exe dekker også mer, og et ”assembly” i .NET dekker enda mer)

Oversikt (se figurene)

- Man kan godt generere utførbar maskin-kode for gitt maskin direkte fra syntaks-treet
  - Angitt som piler mot *høyre* på figur II
  - Bruker da altså ikke mellomkode (eller bare *veldig maskinnær* mellomkode).
  - Genereres tradisjonelt på én av tre former (ovenfra og ned på figuren):

  (1) Maskinkode på tekstlig assembly-format (som så gjøres om til format 2 vha. en assembler).

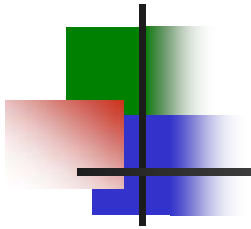
  (2) Vanligst: Som ”relokerbart format” som viderebehandles av tradisjonell loader til format 1

  (3) Direkte som binær maskin-kode (som ev. kan legges rett ned i maskinen)

- Men: Det kan også være greit å overføre programmet til en halvkompilert form, som ikke er avhengig av noen spesiell maskin:
  - Kalles ofte ”mellom-kode”. To typer (som glir over i hverandre f.eks. ved JIT-kompilering):

    (1) Intern (tradisjonell) mellomkode, for intern bruk i kodegeneratoren

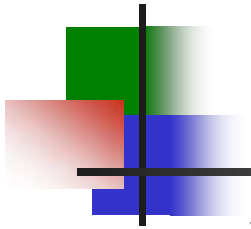
    (2) Utførbar mellomkode («byte-kode» som ofte også blir kompilert videre til maskinkode)
  - Disse formene er nokså like, og vi skal her snakke om begge typer samlet (men i det vi tar med fra kap. 8 snakker boka mest om den *interne* typen)



8.1 Bruk av mellomkode

- Vi skal se på to ”stilarter” for slik mellom-kode:
  - Treadresse-kode (TA-kode)
    - Setter nye navn på alle mellomresultater (kan tenkes på som registre av ubegrenset antall)
    - Fordel: Forholdsvis lett å snu om på rekkefølgen av instruksjonene (for optimalisering)
  - P-kode (Pascal-kode – a la Javas ”byte-kode”, og den til Oblig 2!)
    - Var opprinnelig beregnet på interpretering, men oversettes nå gjerne
    - Mellomresultatene på en stakk (operasjonene komme postfiks)

- Mange valg for detaljer i begge stilarter, f.eks.:
  - Er adresser oversatt til tall, eller er de på tekstlig form?
  - Er det egne operasjoner for f.eks. array-aksess, eller blir slike større operasjoner delt opp i flere og mer elementære instruksjoner?
  - Er det angivelse av typer på operandene?



Vi skal se på en del oversettelser:

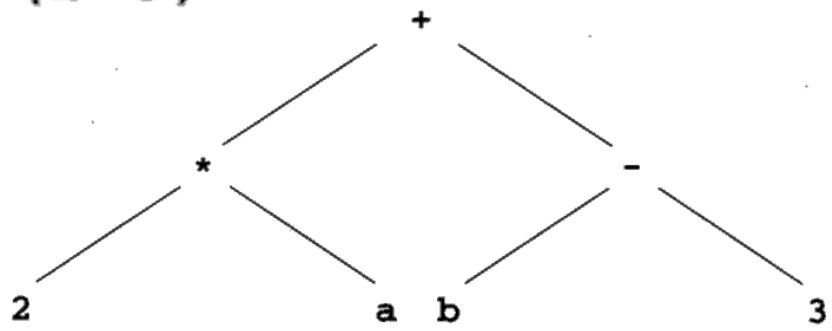
- Vi skal først se på:
  - Generering av TA-kode ut fra tre-strukturen fra sem. analyse
  - Generering av P-kode ut fra tre-strukturen fra sem. analyse
    - Dette er omtrent som i Oblig 2
  - Generering av TA-kode fra P-kode
  - Generering av P-kode fra TA-kode

- Vi kommer da på veien borti mange av problemene ved generell kodegenerering

- Men vi kommer *ikke* borti "registerallokering":
  - Altså: Hvor skal vi holde dataene for at de til enhver tid skal være raskest mulig å få tak i (i registre er raskest, men det er gjerne forholdsvis få registre)
  - Denne problemstillingen er blitt litt "forkludret" etter at det ble vanlig med cacher, kanskje med flere nivåer
  - Vi skal se eksplisitt på registerallokering senere

Tre-adresse (TA)-kode - eksempel

```
2*a+(b-3)
```



TA grunnform:

$$x = y \; op \; z$$

op = +, - , *, /, < , > ,
and, or, ...

Tre-adresse (TA) kode

```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

Også:

x = op y

op = not, -, float-to-int. ...

En alternativ kode-sekvens

```
t1 = b - 3
t2 = 2 * a
t3 = t2 + t1
```

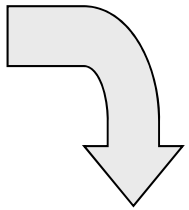
Andre TA-instruksjoner:

x = y
if_false x goto L
label L (”psudo-instr.”)
read x
write x

t_1, t_2, t_3,... er temporære variable (kan innføre stadig nye).

”Hånd”-oversettelse til treadresse-kode

```
1  read x; { input an integer }
2  if 0 < x then { don't compute if x <= 0 }
3    fact := 1;
4    repeat
5      fact := fact * x;
6      x := x - 1
7    until x = 0;
8    write fact { output factorial of x }
9  end
```



```
1  read x
2  t1 = x > 0
   if_false t1 goto L1
3  fact = 1
4  label L2
5  t2 = fact * x
   fact = t2
6  t3 = x - 1
   x = t3
7  t4 = x == 0
   if_false t4 goto L2
8  write fact
   label L1
9  halt
```

Eller ?:
x = x - 1

Mange valg å gjøre ved design av TA-kode:

-Er det egne instruksjoner for int, long, float,..?

-Hvordan er variable representert?

- -ved navn?
- -peker til deklarasjon i symbol-tabell?
- -ved maskinadresse?

-Hvordan er hver instruksjon lagret?

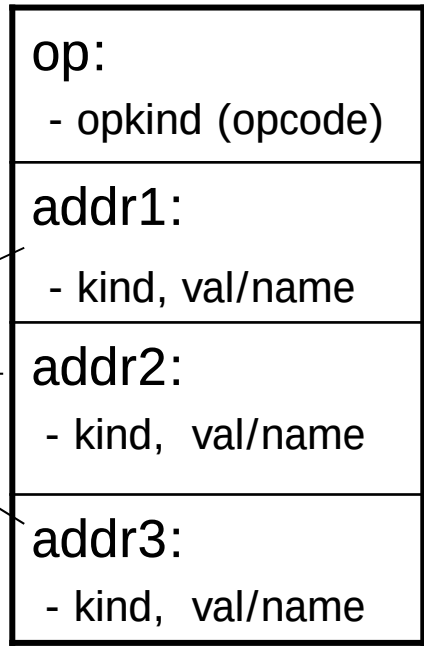
- - kvadrupler, de tre adressene, og operasjonen
- - (Eller: tripler, der ”adressen” til instruksjonen er navn på en ny temporær variabel)

En mulig C-struct for å lagre en treadresse-instruksjon

operasjonskodene

```
typedef enum {rd,gt,if_f,asn,lab,mul,
                    sub,eq,wri,halt,. . .} OpKind;
typedef enum {Empty,IntConst,String} AddrKind;
typedef struct
            { AddrKind kind;
              union
              { int val;
                char * name;
              } contents;
            } Address;
typedef struct
            { OpKind op;
              Address addr1,addr2,addr3;
            } Quad;
```

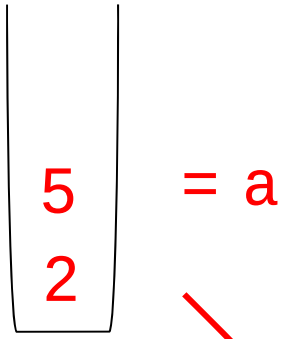
Hver adresse har denne formen



P-kode (opprinnelig for Pascal) utfører beregning på en stakk
Instruksjonene utføres normalt etter hverandre, unntak: jump

"push" instr. (= load til stakken):

```
lod   ;  load value (burde jo vært ldv!)
ldc   ;  load constant
lda   ;  load address
```



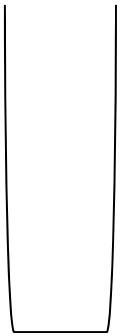
`2*a+(b-3)`

```
ldc 2          ; load constant 2
lod a          ; load value of variable a
mpi            ; integer multiplication
lod b          ; load value of variable b
ldc 3          ; load constant 3
sbi            ; integer subtraction
adi            ; integer addition
```

P-kode II

```
x := y + 1
```

```
lda x          ; load address of x
lod y          ; load value of y
ldc 1          ; load constant 1
adi            ; add
sto            ; store top to address
               ; below top & pop both
```



P-kode for fakultets-funksjonen

Blir typisk mange flere P-instruksjoner enn TA-instruksjoner for samme program
(Hver P-instruksjon har maks én ”lager-adresse”)

```
1  read x; { input an integer }
2  if 0 < x then { don't compute if x <= 0 }
3    fact := 1;
4    repeat
5      fact := fact * x;
6      x := x - 1
7    until x = 0;
8    write fact { output factorial of x }
9  end
```

```
1  lda x        ; load address of x
   rdi          ; read an integer, store to
                ; address on top of stack (& pop it)
2  lod x        ; load the value of x
   ldc 0        ; load constant 0
   grt          ; pop and compare top two values
                ; push Boolean result
   fjp L1       ; pop Boolean value, jump to L1 if false
3  lda fact     ; load address of fact
   ldc 1        ; load constant 1
   sto          ; pop two values, storing first to
                ; address represented by second
4  lab L2       ; definition of label L2
5  lda fact     ; load address of fact
   lod fact     ; load value of fact
   lod x        ; load value of x
   mpi          ; multiply
   sto          ; store top to address of second & pop
6  lda x        ; load address of x
   lod x        ; load value of x
   ldc 1        ; load constant 1
   sbi          ; subtract
   sto          ; store (as before)
7  lod x        ; load value of x
   ldc 0        ; load constant 0
   equ          ; test for equality
   fjp L2       ; jump to L2 if false
8  lod fact     ; load value of fact
   wri          ; write top of stack & pop
   lab L1       ; definition of label L1
9  stp
```

Angivelse av P-kode ved attr.-grammatikk.

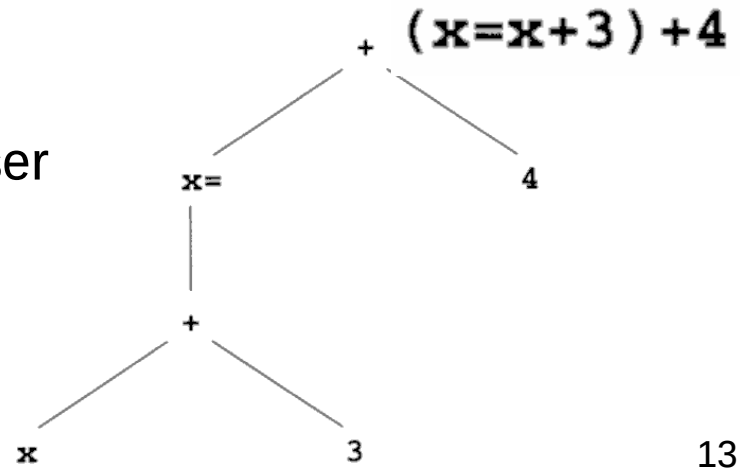
NB: Alle noder har tekst-attributtet «pcode», bortsett fra ***id***, som har «strval»

Grammar Rule	Semantic Rules
$exp_1 \rightarrow$ **id** = exp_2	exp_1 .pcode = "**lda**" || **id** .strval ++ exp_2 .pcode ++ "**stn**"
$exp \rightarrow aexp$	exp .pcode = aexp .pcode
$aexp_1 \rightarrow aexp_2$ + factor	$aexp_1$ .pcode = $aexp_2$ .pcode ++ factor .pcode ++ "**adi**"
$aexp \rightarrow factor$	aexp .pcode = factor .pcode
$factor \rightarrow$ (exp)	factor .pcode = exp .pcode
$factor \rightarrow$ **num**	factor .pcode = "**ldc**" || **num** .strval
$factor \rightarrow$ **id**	factor .pcode = "**lod**" || **id** .strval

|| betyr: sette sammen til én instruksjon

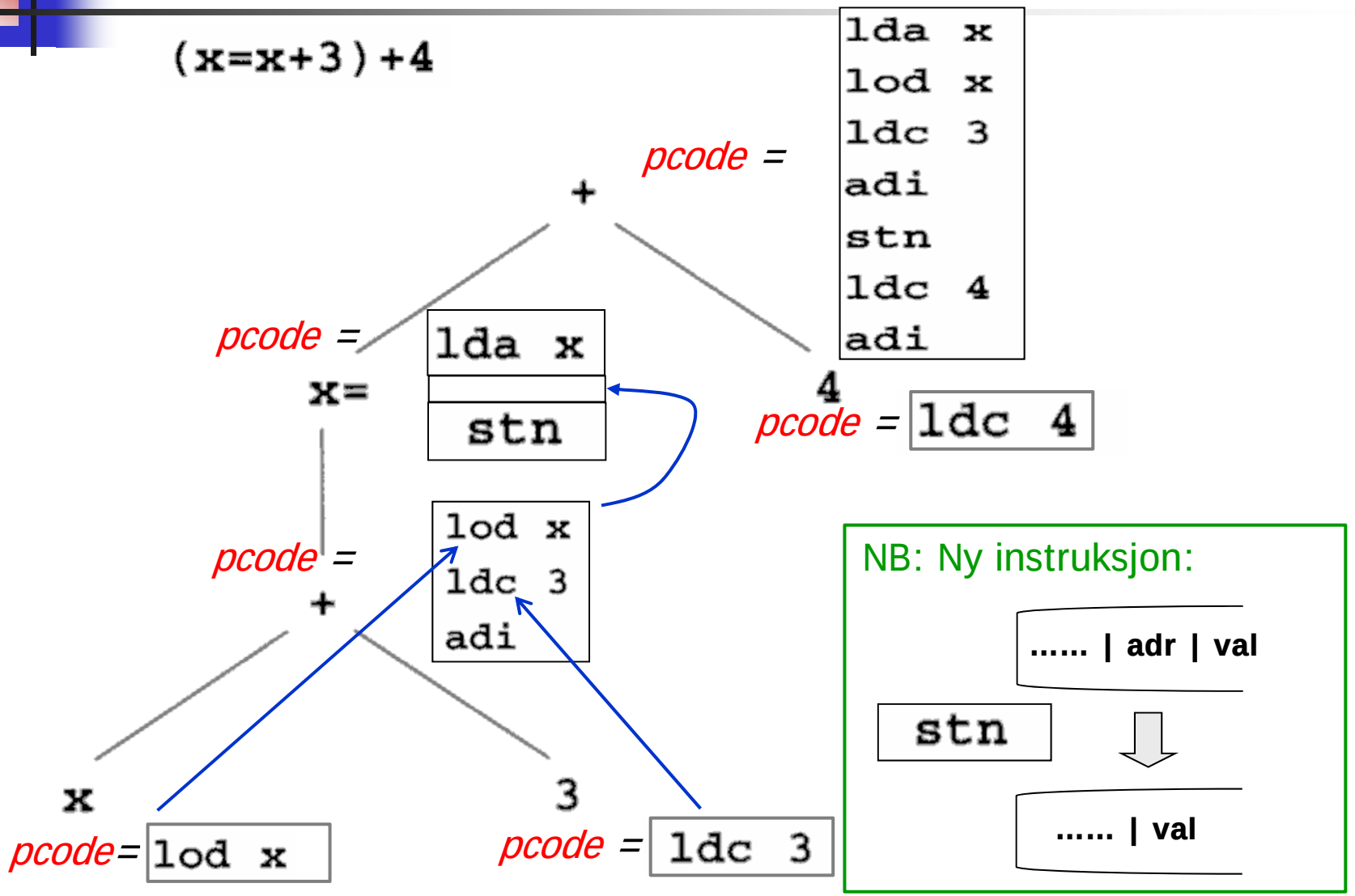
++ betyr: sette sammen instruksjonsekvenser

NB: Kodegenerering direkte ut fra denne er *veldig* upraktisk!



Tenkt generering av P-kode etter attr.-gram.

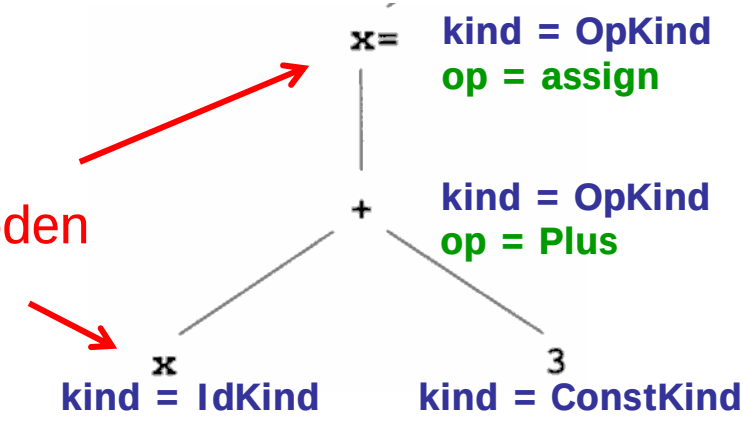
(Vil altså aldri gjøre det slik i praksis!)



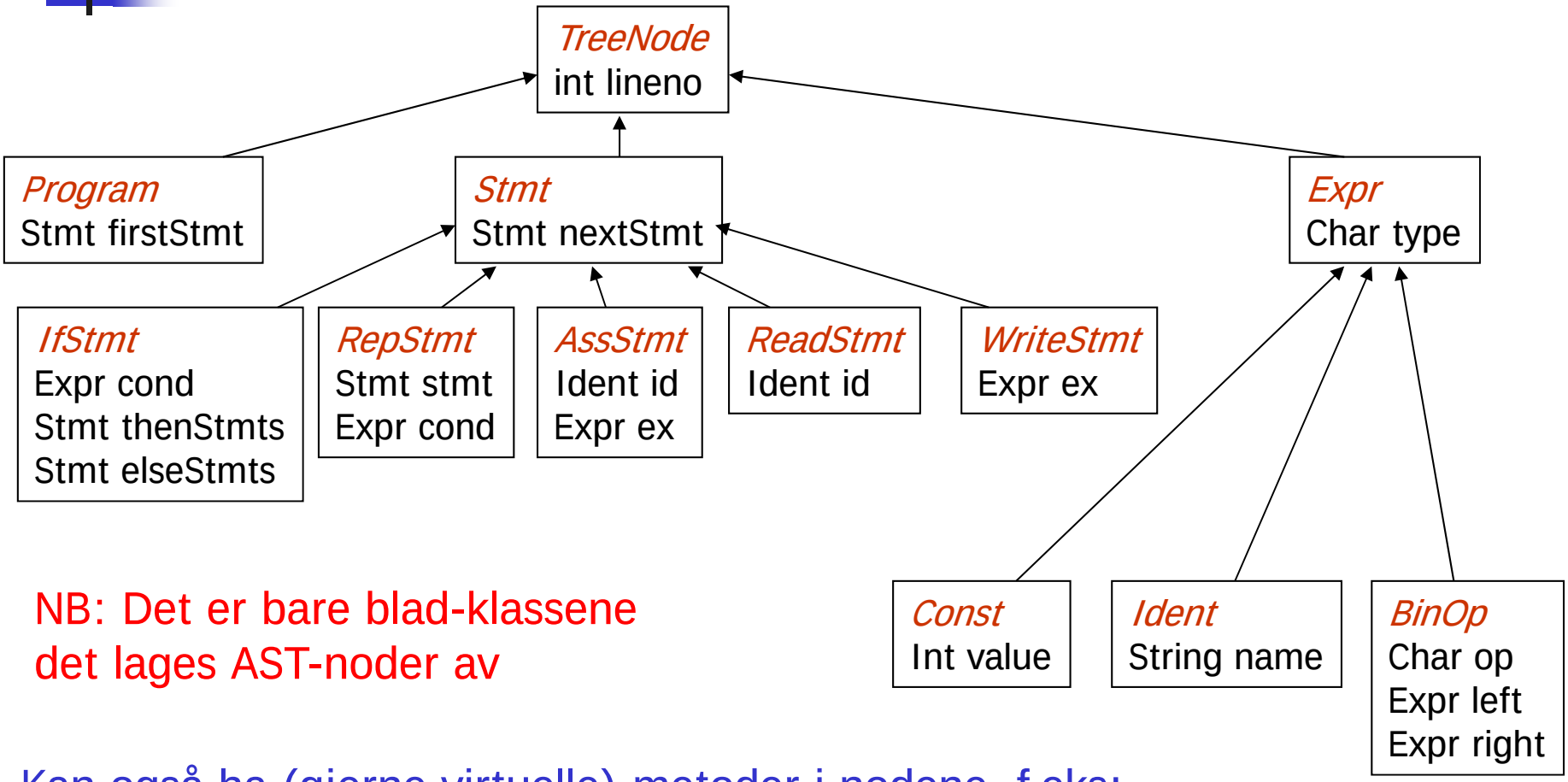
Slik kodegenerering kan bedre gjøres ved rekursiv gjennomgang av syntakstreet. Forslag til tre-node:

```
typedef enum {Plus,Assign} Optype;
typedef enum {OpKind,ConstKind,IdKind} NodeKind;
typedef struct streenode
        { NodeKind kind;
          Optype op; /* used with OpKind */
          struct streenode *lchild,*rchild;
          int val; /* used with ConstKind */
          char * strval;
            /* used for identifiers and numbers */
        } STreeNode;
typedef STreeNode *SyntaxTree;
```

Navnet x ligger i noden



Minner om en annen, mer objektorientert, måte enn å bare ha én node-klasse som dekker alle behov: Vi bruker f.eks. følgende hierarki av klasser som beskriver nodene i treet (AST)



NB: Det er bare blad-klassene det lages AST-noder av

Kan også ha (gjerne virtuelle) metoder i nodene, f.eks:

- doSemAnalyses(); Gjør semantisk analyse av noden, og av subtreet det er rot i
- generateCode(); Genererer kode for noden, og for subtreet det er rot i

Metode-skisse til generelt bruk

ved rekursiv traversering av trær

```
procedure genCode ( T: treenode );
begin
   if T is not nil then
      generate code to prepare for code of left child of T ;
      genCode(left child of T) ;
      generate code to prepare for code of right child of T ;
      genCode(right child of T) ;
      generate code to implement the action of T ;
end;
```

← **Prefiks - operasjoner**

← **rekursivt kall**

← **Infiks - operasjoner**

← **rekursivt kall**

← **Postfiks - operasjoner**

Generert kode for hele subtreet (oftest er flere av delene tomme):

oppstarts-kode	bergen v. operand	fiks på v. operand	beregn h. operand	gjør operasjonen

Boka: Generering av P-kode fra AST. Alle nodene samme metode

```
void genCode( SyntaxTree t)
{ char codestr[CODESIZE];
  /* CODESIZE = max length of 1 line o
  if (t != NULL)
  { switch (t->kind)
    { case OpKind:
        switch (t->op)
        { case Plus:
            genCode(t->lchild);   ← rek.kall
            genCode(t->rchild);   ← rek.kall
            emitCode("adi");
            break;
```

Oversiktlig versjon:

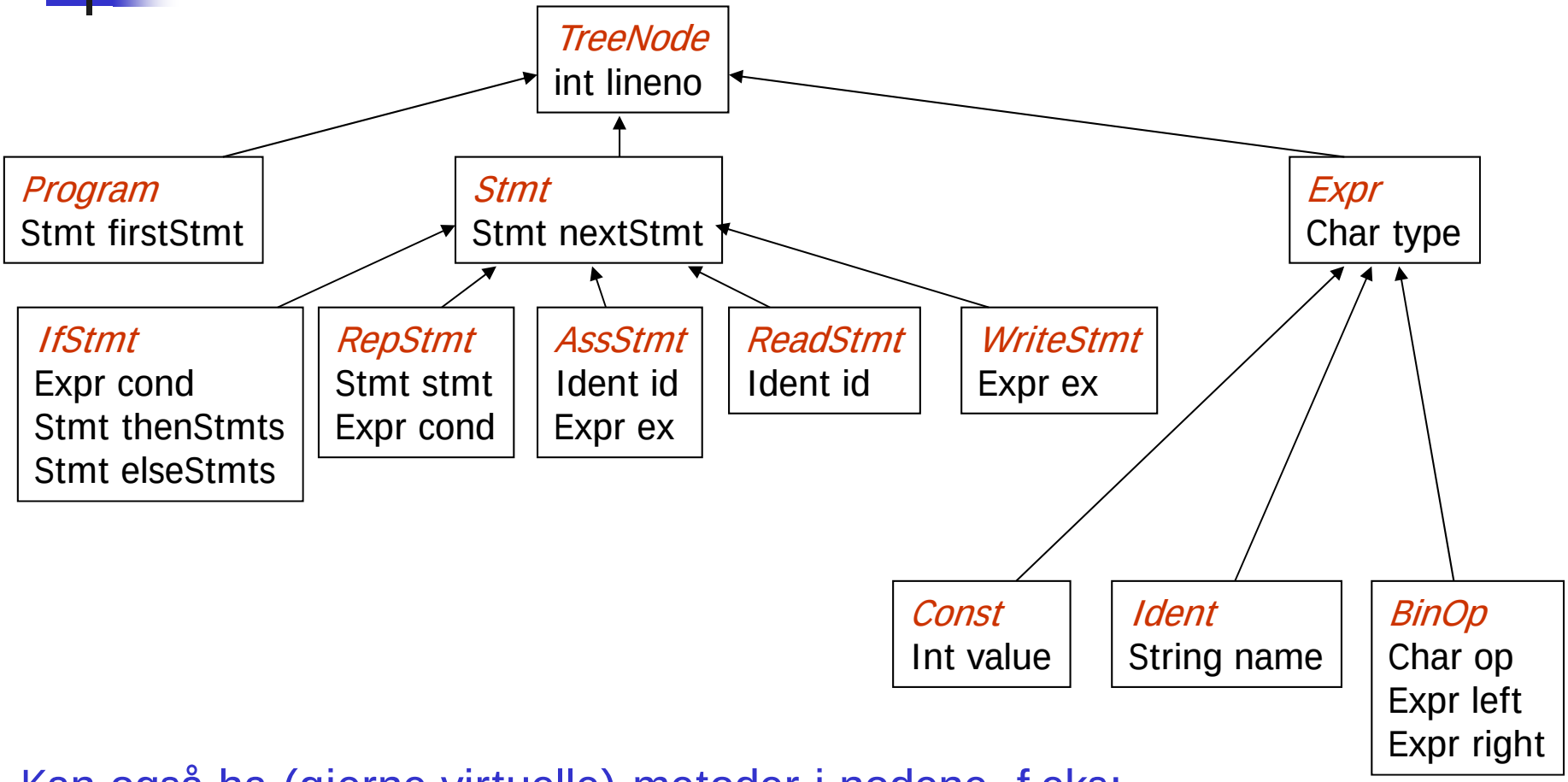
```
switch kind {
  case OpKind:
    switch op{
      case  Plus: { rek. kall for venstre subtre;
                    rek. kall for høyre subtre;
                    emit1 (”adi”); }
      case  Assign: { emit2 (”lda”, identifikator);
                      rek. kall for eneste subtre;
                      emit1 (”stn”); }
    }
  case ConstKind { emit2 (”ldc”, konstant-streng); }
  case IdKind  { emit2 (”lod”, identifikator); }
}
```

Merk: Identifikator og konstant-streng ligger i noden

```
        case Assign:
          sprintf(codestr,"%s %s",
                          "lda",t->strval);
          emitCode(codestr);
          genCode(t->lchild);     ← rek.kall
          emitCode("stn");
          break;
        default:
          emitCode("Error");
          break;
      }
      break;
    case ConstKind:
      sprintf(codestr,"%s %s","ldc",t->strval);
      emitCode(codestr);
      break;
    case IdKind:
      sprintf(codestr,"%s %s","lod",t->strval);
      emitCode(codestr);
      break;
    default:
      emitCode("Error");
      break;
  }
}
}
```

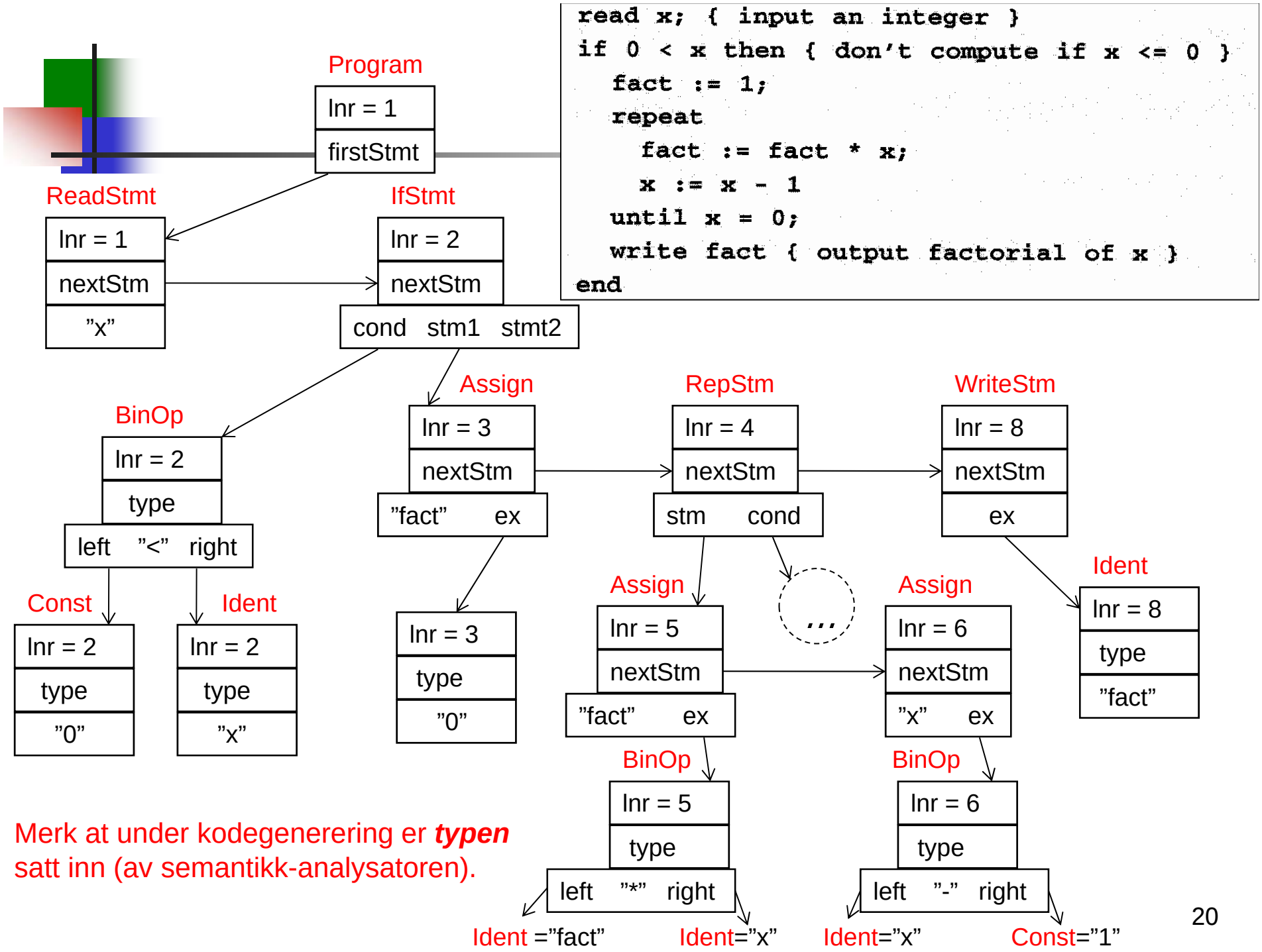
Nodeklasser for OO-utgave av abst.-synt.-tre for Tiny-språket.

Merk: Dette er altså en fast subklasse-struktur i kompilatoren, og må ikke forveksles med det abstrakte syntaks-treet for et gitt program!

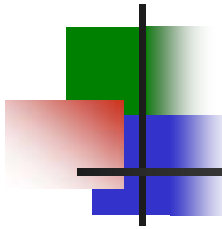


Kan også ha (gjerne virtuelle) metoder i nodene, f.eks:

- doSemAnalyses(); Gjør semantisk analyse av noden, og av subtreet det er rot i
- generateCode(); Genererer kode for noden, og for subtreet det er rot i



Merk at under kodegenerering er ***typen*** satt inn (av semantikk-analysatoren).



Det er her greiest med egne «override-metoder» i hver node-klasse (som i Oblig2). Her for yttrykk

Ser på void GenCode() {...} for å lage P-kode

- Vi lager først en abstrakt void GenCode(); i klassen Expr (eller gjerne i klassen TreeNode).
- Så en konkret versjon i hver av blad-klassene Const, Ident og BinOp:

- I klassen Const:

```
void GenCode() { emit ("ldc " + Value); }
```

- I klassen Ident:

```
void GenCode() { emit ("lod " + Name); }
```

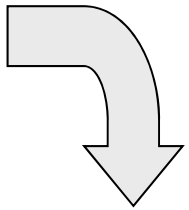
- I klassen BinOp:

```
void GenCode(){
    left.GenCode();   // Kan være et større uttrykks-tre
    right.GenCode(); //              ---- " ----
    emit ( <P-kode-instruksjonen tilsv. op> );
}
```

”Hånd”-oversettelse til treadresse-kode g

(Vist tidligere)

```
1  read x; { input an integer }
2  if 0 < x then { don't compute if x <= 0 }
3    fact := 1;
4    repeat
5      fact := fact * x;
6      x := x - 1
7    until x = 0;
8    write fact { output factorial of x }
9  end
```



```
1  read x
2  t1 = x > 0
   if_false t1 goto L1
3  fact = 1
4  label L2
5  t2 = fact * x
   fact = t2
6  t3 = x - 1
   x = t3
7  t4 = x == 0
   if_false t4 goto L2
8  write fact
   label L1
9  halt
```

Eller ?:
x = x - 1

Mange valg å gjøre ved design av TA-kode:

-Er det egne instruksjoner for int, long, float,..?

-Hvordan er variable representert?

- -ved navn
- -peker til deklarasjon i symbol-tabell
- -ved maskinadresse

-Hvordan er hver instruksjon lagret?

- - kvadrupler, de tre adressene, og operasjonen
- - (Eller: tripler, der ”adressen” til instruksjonen er navn på en ny temporær variabel)

Angivelse av TA-kode ved attr.-grammatikk

NB: Har metoden ”String newTemp()” som skaffer nytt, ubrukt navn til temporær variabel

```
(x=x+3)+4
```

NB: Alle noder har *to* attributter:

name:
Navnet på den variablen der svaret for dette subtreet ligger

tacode:
All TA-koden for hele subtreet

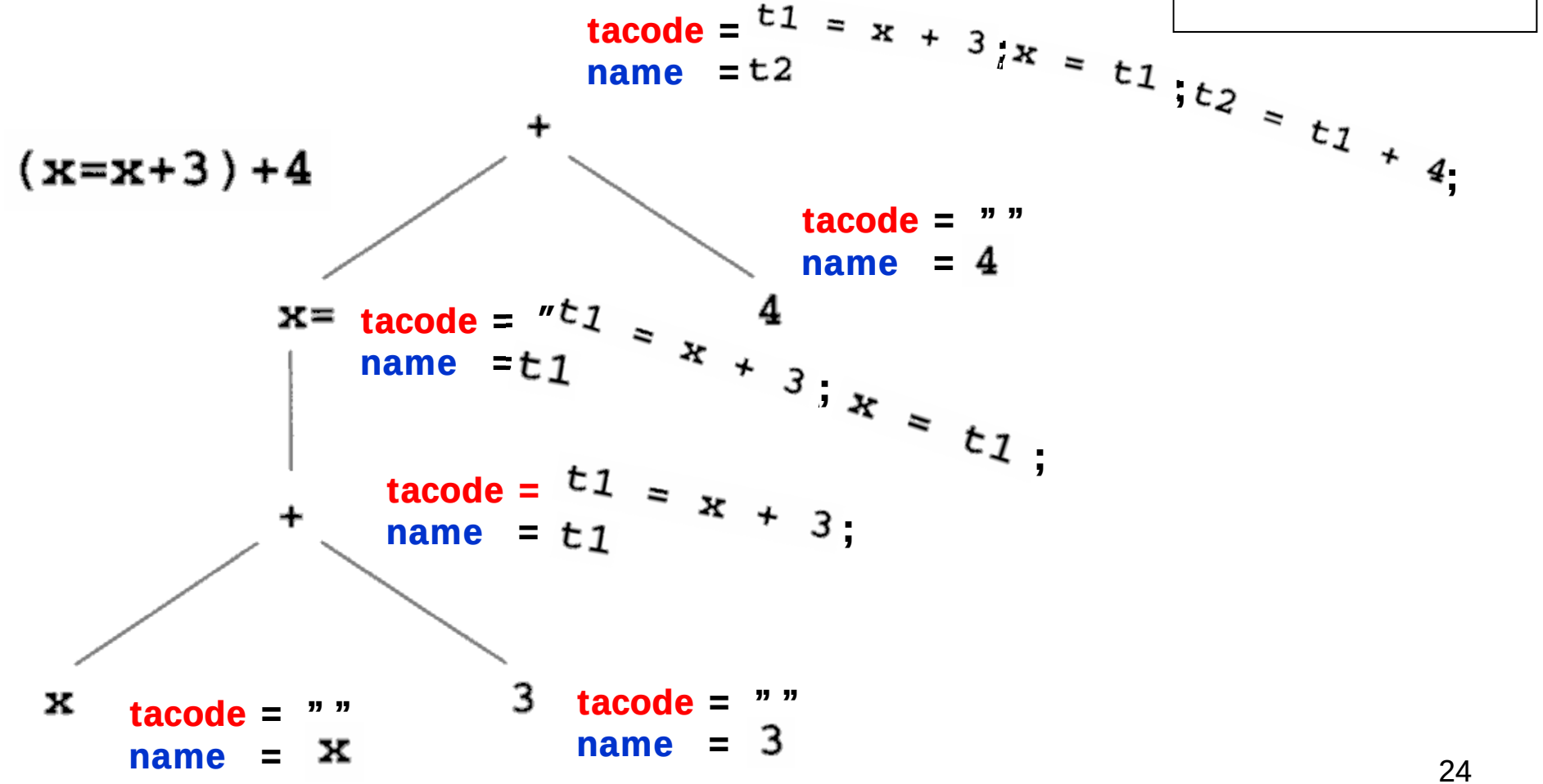
Grammar Rule	Semantic Rules
$exp_1 \rightarrow$ **id** = exp_2	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode$ ++ **id**.strval ‖ “=” ‖ $exp_2.name$
$exp \rightarrow aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \rightarrow aexp_2$ + $factor$	$aexp_1.name = newtemp()$ $aexp_1.tacode =$ $aexp_2.tacode$ ++ $factor.tacode$ ++ $aexp_1.name$ ‖ “=” ‖ $aexp_2.name$ ‖ “+” ‖ $factor.name$
$aexp \rightarrow factor$	$aexp.name = factor.name$ $aexp.tacode = factor.tacode$
$factor \rightarrow$ (exp)	$factor.name = exp.name$ $factor.tacode = exp.tacode$
$factor \rightarrow$ **num**	$factor.name =$ **num**.strval $factor.tacode =$ ""
$factor \rightarrow$ **id**	$factor.name =$ **id**.strval $factor.tacode =$ ""

Tenkt generering av TA-kode etter attr.-gram.

(Gjøres ikke slik i praksis)

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

”name”: navn på variabelen der svaret ligger



Bokas generering av TA-kode fra AST. Ser på uttryks-noder-

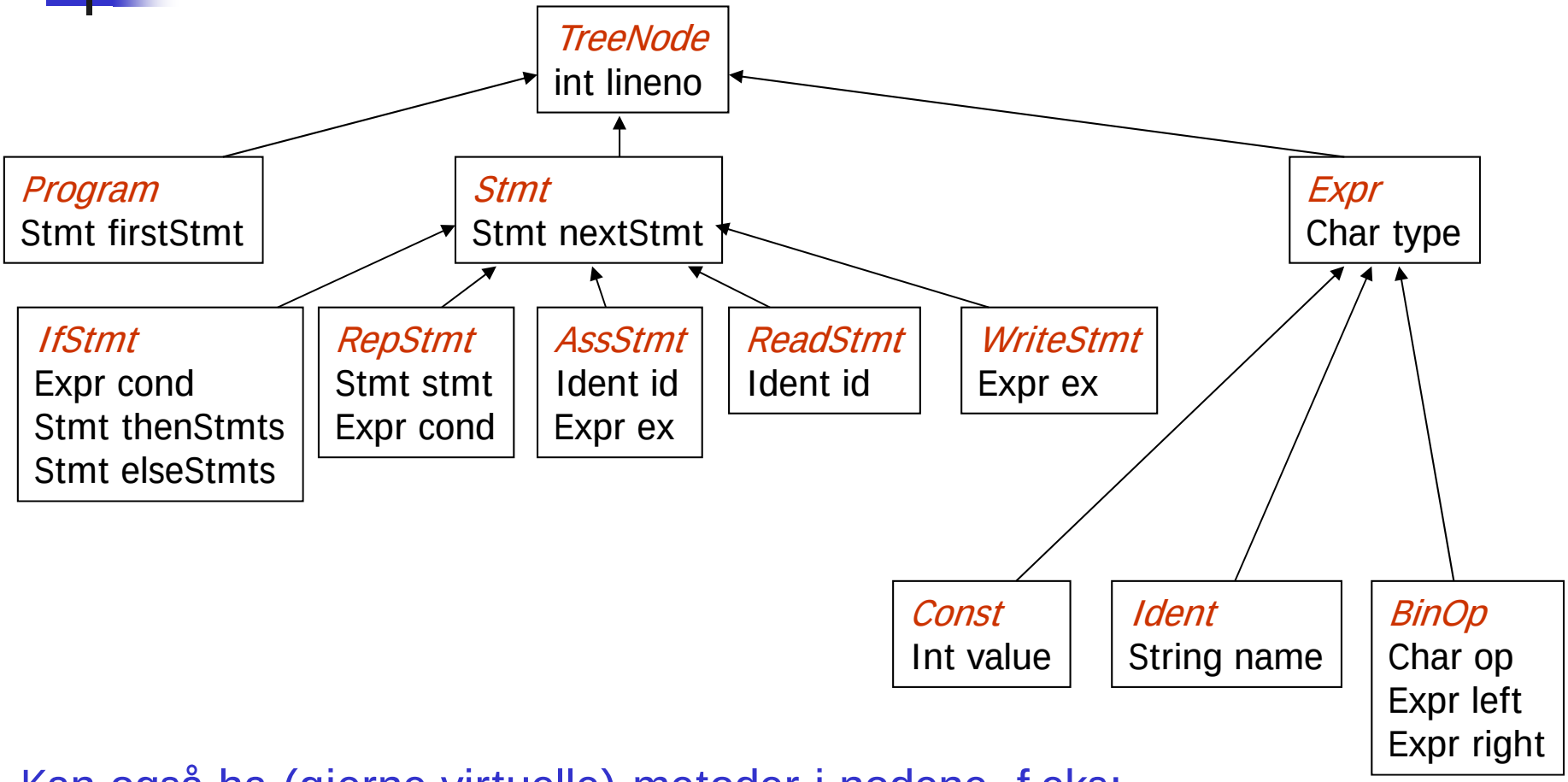
Bruker en rekursiv metode, som her er lik i alle noder:

Metoden under eksekverer inne i nodene, lager kode, og leverer navnet (en String») på variabelen der verdien av uttrykket vil ligge når den produserte kode er utført.

```
switch kind {
  case OpKind:
    switch op {
      case Plus: { tempnavn = nytt temporær-navn;
                   varnavn1 = rek kall for venstre subtre;
                   varnavn2 = rek kall for høyre subtre;
                   emit (”tempnavn = varnavn1 + varnavn2”);
                   return (tempnavn); }
      case Assign: { varnavn = id. for v.s.-variabel (ligger i noden);
                     varnavn1 = rek kall for venstre subtre;
                     emit (”varnavn = opnavn”);
                     return (varnavn); }
    }
  case ConstKind: { return (konstant-streng); }  // ”Emitter” ingenting!
  case IdKind:      { return (identifikator); }       // ”Emitter” ingenting!
}
```

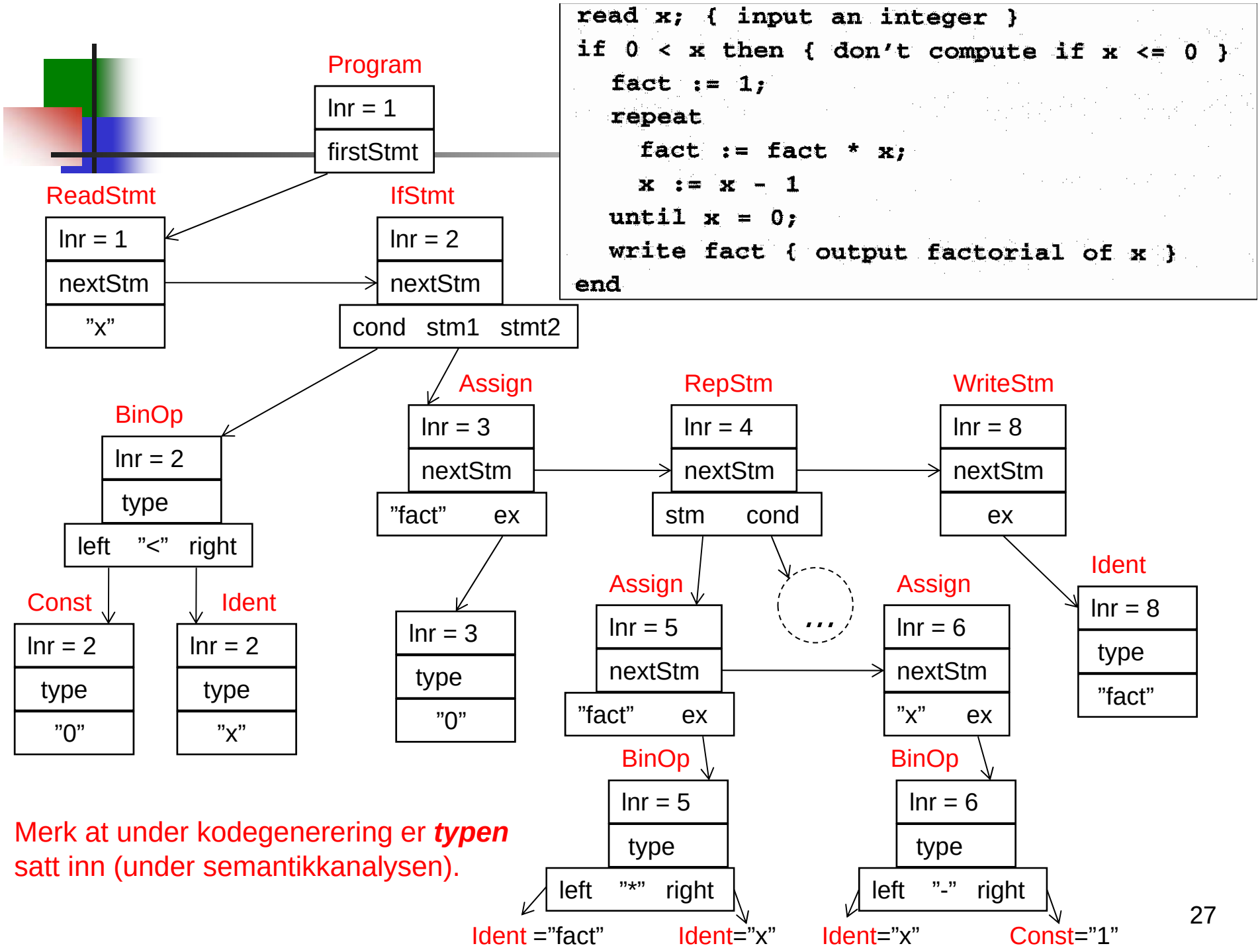
Nodeklasser for OO-utgave av abst.-synt.-tre for Tiny-språket.

Merk: Dette er altså en fast subklasse-struktur i kompilatoren, og må ikke forveksles med det abstrakte syntaks-treet for et gitt program!



Kan også ha (gjerne virtuelle) metoder i nodene, f.eks:

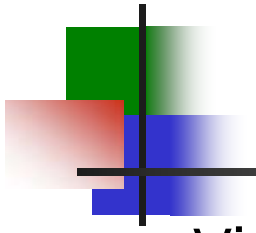
- doSemAnalyses(); Gjør semantisk analyse av noden, og av subtreet det er rot i
- generateCode(); Genererer kode for noden, og for subtreet det er rot i



Merk at under kodegenerering er ***typen*** satt inn (under semantikkanalysen).

Skjema for «String GenCodeTA()»

(Vi har en metode «String NewTemp()» som leverer nye ubrukte navn)



- Vi lager først en abstrakt «String GenCodeTA();» i klassen Expr (eller gjerne i klassen TreeNode).
- Deretter en konkret variant i hver av node-klassene Const, Ident og BinOp:

- I klassen Const:

```
String GenCodeTA() {  ???  }
```

I klassen Ident:

```
String GenCodeTA() {  ???  }
```

- I klassen BinOp:

```
String GenCodeTA(){
    ??
    ??
    ??
}
```

Svarforslag for «String GenCodeTA()»

(vi har en metode String NewTemp() som leverer nye ubrukte navn)

- Vi lager først en abstrakt «String GenCodeTA();» i klassen Expr (eller gjerne i klassen TreeNode).
- Deretter en konkret variant i hver av node-klassene Const, Ident og BinOp:

- I klassen Const:

```
String GenCodeTA() { return "" + value; } // Konst. som String
```

I klassen Ident:

```
String GenCodeTA() {return name; } // Verdien i denne variablen
```

- I klassen BinOp:

```
String GenCodeTA(){   String s1, s2;  String t = NewTemp();
      s1 = left.GenCodeTA();       // Kan være et helt uttrykks-tre
      s2 = right.GenCodeTA();      //          ... samme ...
      emit ( t + " = " + s1 + op +  s2 );
      return  t;
}
```

Fra P-kode til TA-kode (”Statisk simulering”)

Typisk slik det gjøres ved JIT-kompilering av bytekode

```
(x=x+3)+4
```

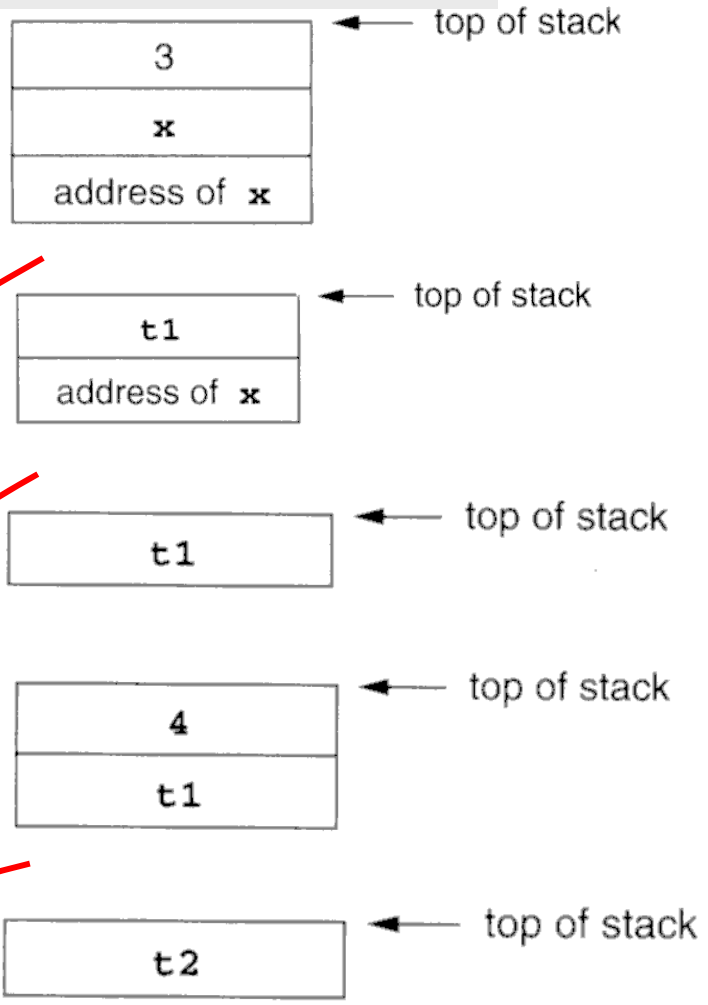
P-kode:

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

Ønskemål:

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

Kompilator-stakkens stadier:



Som vi ser: Vi får den kode-sekvensen vi ønsket oss!

Fra TA-kode til P-kode - ved ”makro-ekspansjon”
Gir vanligvis ikke godt resultat

”Makro” for: a = b + c

```
lda a
lod b ; or ldc b if b is a const
lod c ; or ldc c if c is a const
adi
sto
```

```
(x=x+3)+4
```

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

```
lda t1
lod x
ldc 3
adi
sto
```

```
lda x
lod t1
sto
```

```
lda t2
lod t1
ldc 4
adi
sto
```

Har tidligere sett kortere versjon:

```
(x=x+3)+4
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

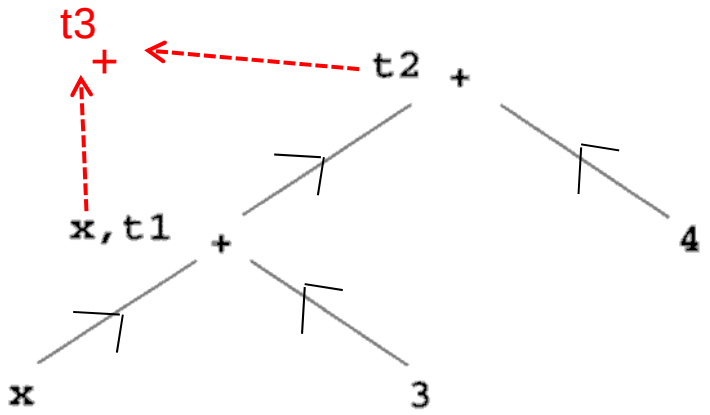
Ny versjon: 13 instr.
Gml. ver. : 7 instr

Fra TA-kode til P-kode: litt lurere, men bare skisse

Prøver å lage bedre kode av:

```
t1 = x + 3
x = t1
t2 = t1 + 4
```

Tegner opp ”data-flyt”-grafen / treet



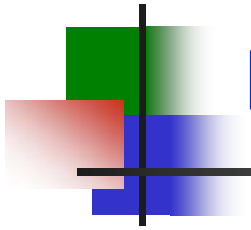
Dette eksempelet blir da et tre, og kan da som før omformes til kodesekvensen under. Trestruktur er altså drømmen for å generere P-kode.

MEN om vi legger til f.eks. instruksjonen: t3 = x + t2
ser vi at dette generelt blir en rettet graf uten løkker (en DAG)
Derved blir det generelt verre.

Men denne grafen kan ofte deles opp i trestrukturer, som man kan bruke som til det er gjort til høyre

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

Og *dette* er jo den optimale som vi startet med!



Lage tradisjonell maskin-kode fra TA-kode

- Dette problemet skal vi se på senere
  - Da vil «registerallokering» bli viktig
    - Hvor skal temporær-verdiene lagres?
  - Også begreperet «Basic block» blir viktig
    - En passelig kode-enhet for å gjøre enkel otimalisering

Kap. 8.3 – Detaljert aksess av datastruktur

Trenger da flere typer instruksjoner til adresse-beregning

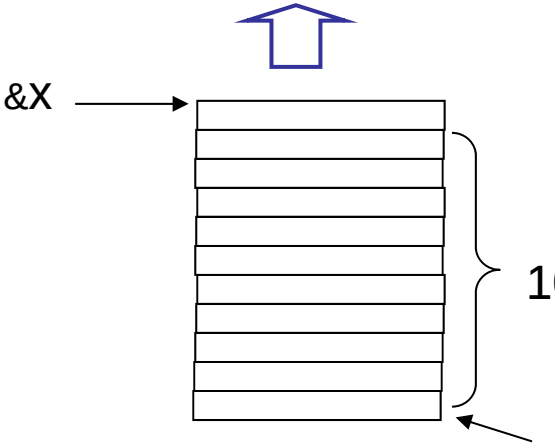
Vil lage kode for: X [10] = 2;

TA-kode: To nye måter å adressere på

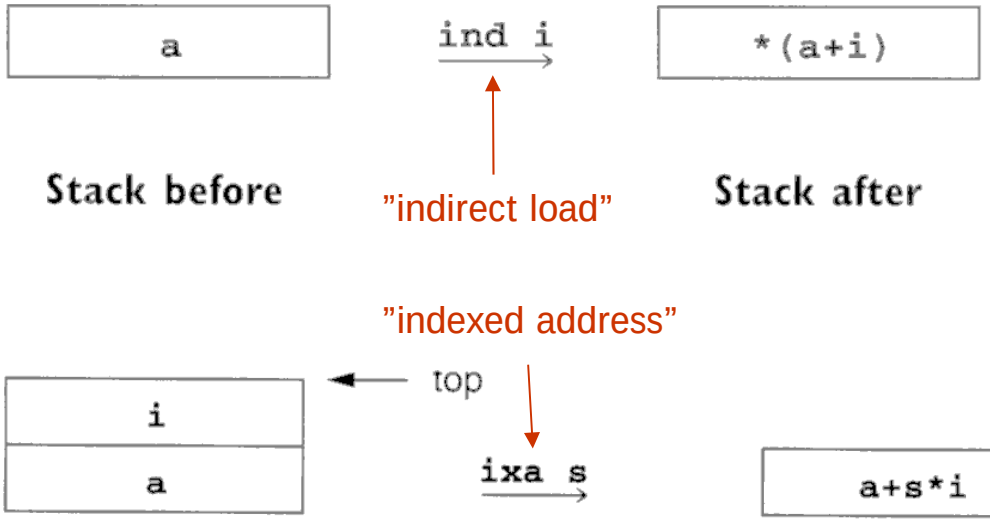
& X Adressen til x (ikke for temporære)

***t** Indirekte gjennom t

```
t1 = &x + 10
*t1 = 2
```



P-kode: To nye instruksjoner

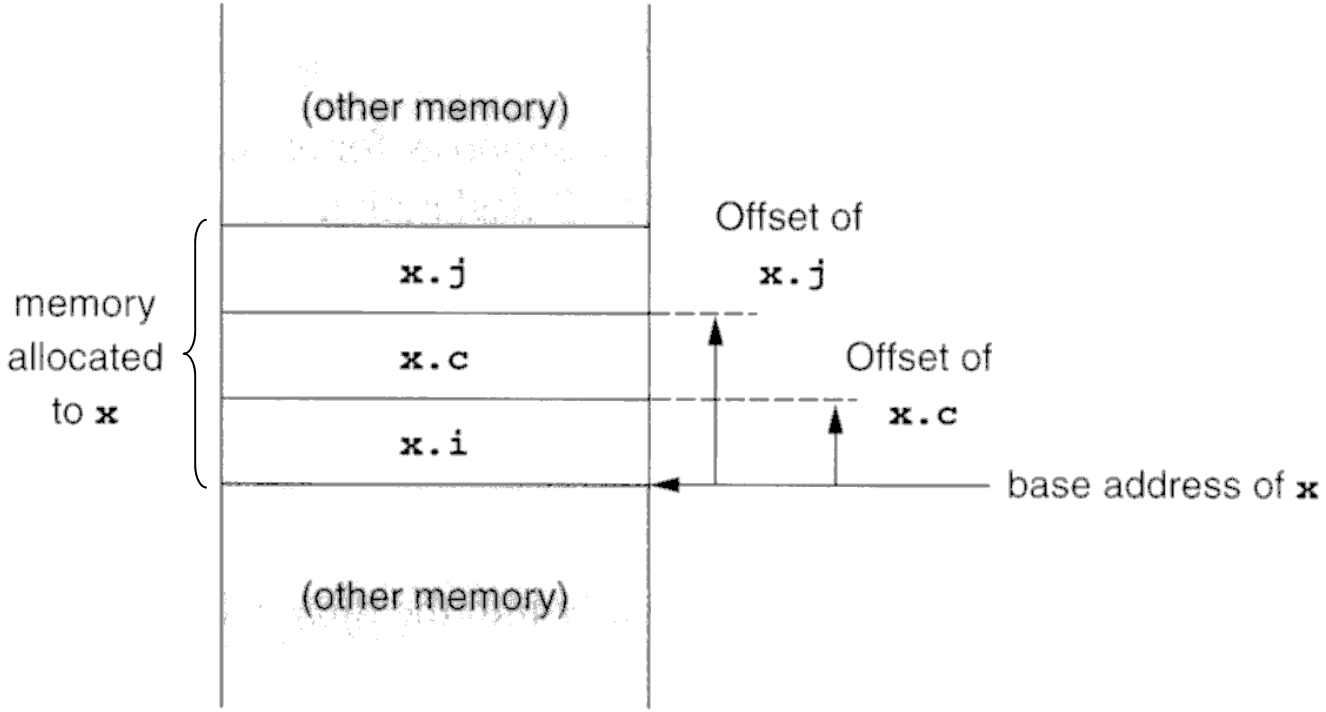


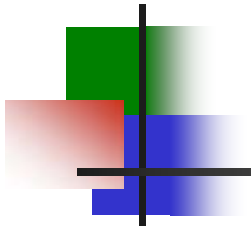

```
lda x
ldc 10
ixa 1
ldc 2
sto
```

Aksessering av data i “structer”, objekter etc.

- Med slike instruksjoner kan vi lage TA-kode og P-kode for å aksessere lokale variable i structer, recorder, objekter etc.
- Vi ser imidlertid ikke på detaljene i dette

```
typedef struct rec
   { int i;
     char c;
     int j;
   } Rec;
...
Rec x;
```





Litt generelt til kap. 8.3

- I boka lages det nokså ”lavnivå” TA-kode og P-kode
  - Ulempe: Man kan da ikke lenger se hva slags språk-konstruksjoner koden kommer fra.
  - Dermed blir det vanskligere å optimalisere på lure steder.

- Det er ikke opplagt at bokas variant er det fornuftigste. Alternativ f.eks. i blokk-strukturerte språk:
  - Beholde aksess til en ikke-lokal variabel på formen:

    X: (rel.niv=2, rel.adr=3)

    I steden for å oversette til formen:

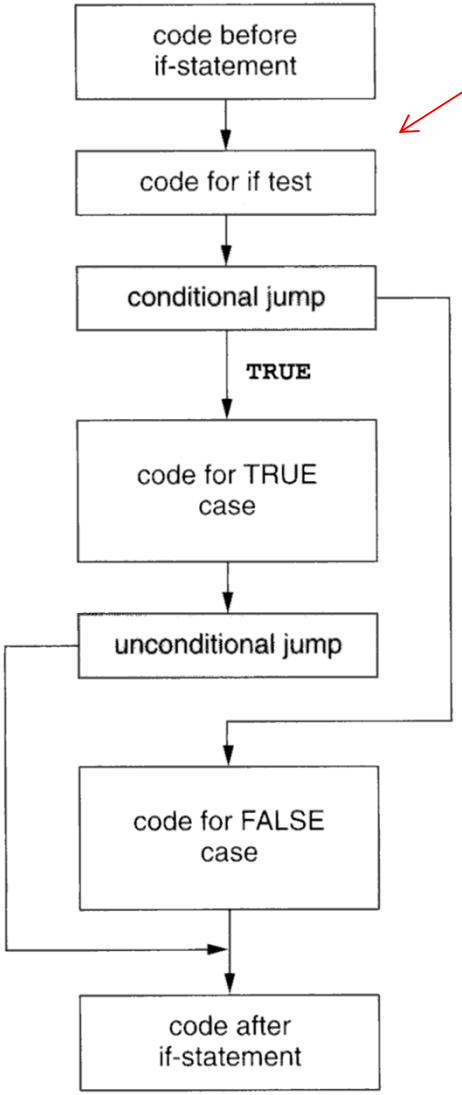
    fp.al.al.(reladr=3)

    som blir et antall i instruksjoner i TA-kode eller P-kode av typen diskutert på forrige foiler.

- Kan gjerne se oversettelse til lav-nivå TA-kode eller P-kode som eksempel på oversettelse direkte til maskin-kode
  - Bortsett fra at vi da slipper register-allokerings-problemet

8.4 : If/while

Kan vi gjøre noe lurere her?
Ja, se senere foiler!



if-stmt → **if** (*exp*) *stmt* | **if** (*exp*) *stmt* **else** *stmt*

while-stmt → **while** (*exp*) *stmt*

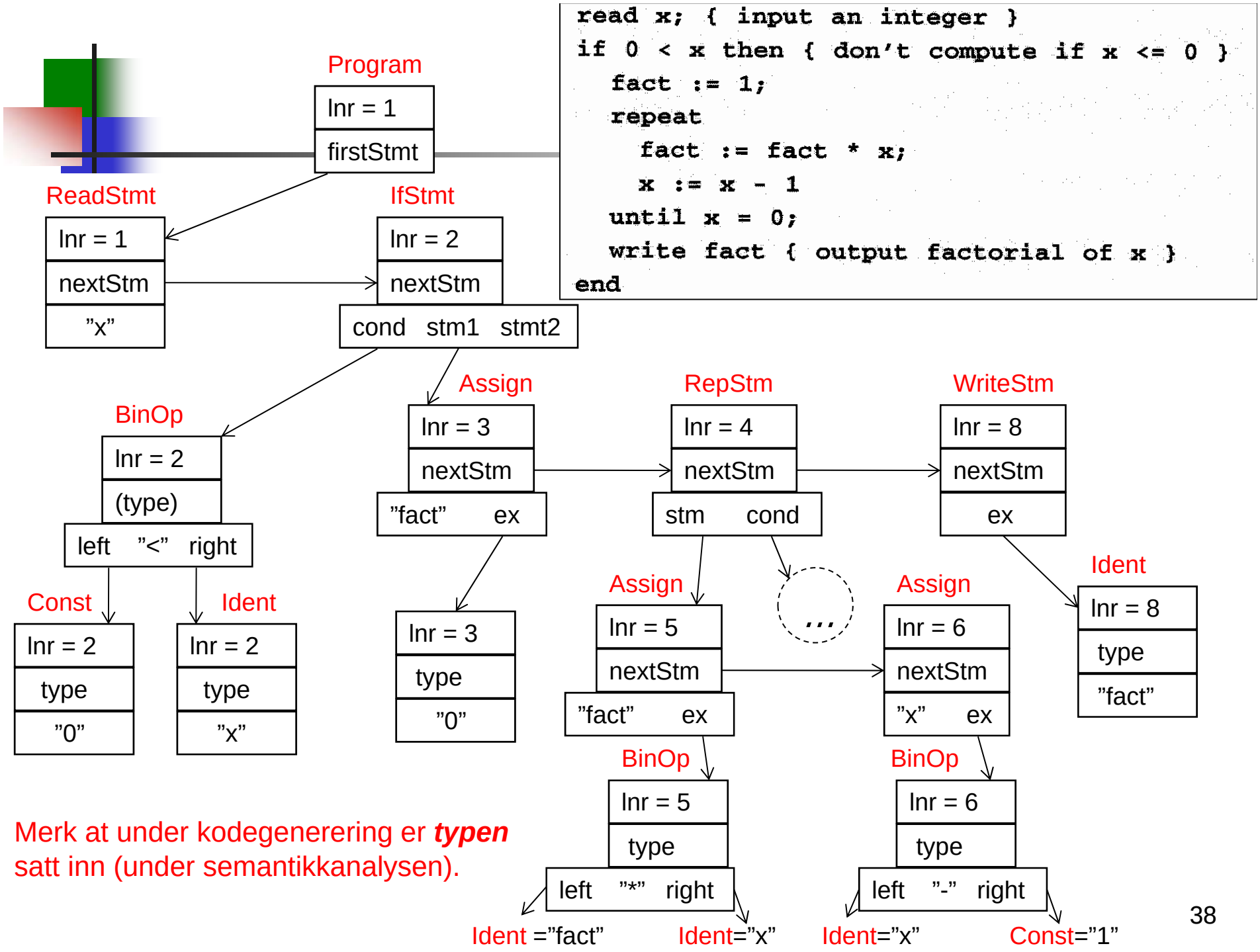
if (*E*) *S1* **else** *S2*

Skisse av TA-kode

```
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
goto L2
label L1
<code for S2>
label L2
```

Skisse av P-kode

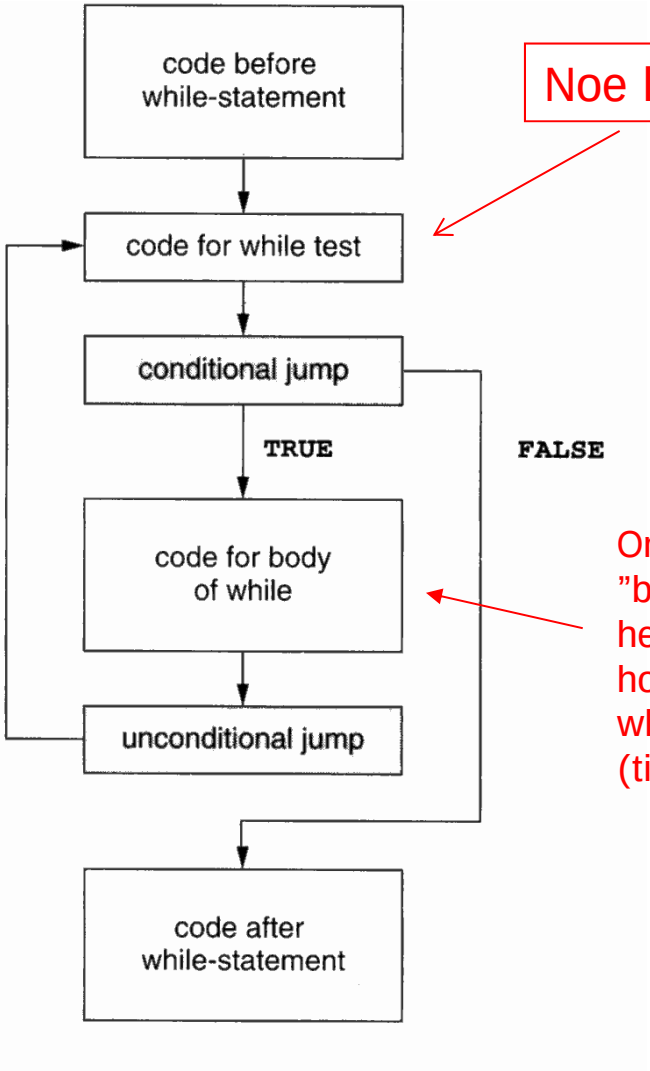
```
<code to evaluate E>
fjp L1
<code for S1>
ujp L2
lab L1
<code for S2>
lab L2
```



Merk at under kodegenerering er ***typen*** satt inn (under semantikkanalysen).

while - setning

while (E) S



Noe lurere her?

Om det skjer et "break" inne i her skal det hoppes ut av while-setningen (til L2)

TA-kode:

```
label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
```

P-kode

```
lab L1
<code to evaluate E>
fjp L2
<code for S>
ujp L1
lab L2
```

Behandling av boolske uttykk

- Mulighet 1: Behandle som vanlige uttrykk
- Mulighet 2: Behandling ved 'kort-slutning'

Men man må følge språkets semantikk!

Eksempel i C – der siste del bare beregnes dersom første del ikke avgjør svaret:

```
if ((p!=NULL) && (p->val==0)) ...
```

$a \text{ and } b \equiv \text{if } a \text{ then } b \text{ else false}$

$a \text{ or } b \equiv \text{if } a \text{ then true else } b$

```
(x!=0)&&(y==x)
```

P-kode:

```
lod x
ldc 0
neq
fjp L1
lod y
lod x
equ
ujp L2
lab L1
ldc FALSE
lab L2
```

a

b

Trykkfeil i boka

<test på om det skal hoppes til else-gren>

Merk:

Om dette uttrykket stod i sammenhengen:

if <utr> then S1 else S2

så kunne hoppet "a" til L1 gå direkte til else – grenen og alt fra og med hoppet b: "ujp L2" kunne fjernes (dog ikke testen under).

Kode for if/while-setninger

stmt → *if-stmt* | *while-stmt* | **break** | **other**
if-stmt → **if(** *exp* **)** *stmt* | **if(** *exp* **)** *stmt* **else** *stmt*
while-stmt → **while(** *exp* **)** *stmt*
exp → **true** | **false**

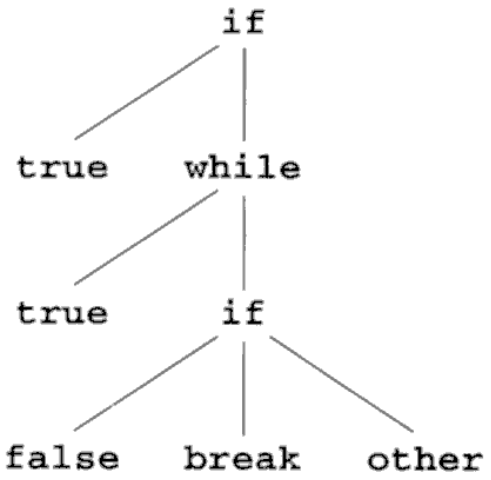
Tre-node:

```
typedef enum {ExpKind,IfKind,
      WhileKind,BreakKind,OtherKind} NodeKind;
typedef struct streenode
        { NodeKind kind;
          struct streenode * child[3];
          int val; /* used with ExpKind */
        } STreeNode;
typedef STreeNode *SyntaxTree;
```

Angir her bare false eller true

```
if(true)while(true)if(false)break else other
```

bindes sammen



Ser noe merkelig ut når alle boolske uttrykk er konstanter

```
ldc true
fjp L1
lab L2
ldc true
fjp L3
ldc false
fjp L4
ujp L3
ujp L5
lab L4
Other
lab L5
ujp L2
lab L3
lab L1
```

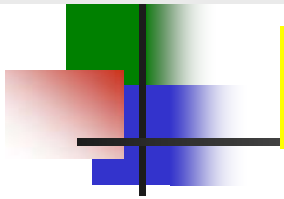
Rekursiv prosedyre for P-kodegenerering for setninger (som i Oblig 2)

En ”break” i kildeprogr. skal bli et hopp til denne labelen

```
void genCode( SyntaxTree t, char * label)
{ char codestr[CODESIZE];
  char * lab1, * lab2;
  if (t != NULL) switch (t->kind)
  { case ExpKind:
      if (t->val==0) emitCode("ldc false")
      else emitCode("ldc true");
      break;
    case IfKind:
      genCode(t->child[0],label);   Rek. kall
      lab1 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab1);
      emitCode(codestr);
      genCode(t->child[1],label);   Rek. kall
      if (t->child[2] != NULL)
      { lab2 = genLabel();
        sprintf(codestr,"%s %s","ujp",lab2);
        emitCode(codestr);}
      sprintf(codestr,"%s %s","lab",lab1);
      emitCode(codestr);
      if (t->child[2] != NULL)
      { genCode(t->child[2],label);   Rek. kall
        sprintf(codestr,"%s %s","lab",lab2);
        emitCode(codestr);}
      break;
```

```
    case WhileKind:
      lab1 = genLabel();
      sprintf(codestr,"%s %s","lab",lab1);
      emitCode(codestr);
      genCode(t->child[0],label);   Rek. kall
      lab2 = genLabel();
      sprintf(codestr,"%s %s","fjp",lab2);
      emitCode(codestr);
      genCode(t->child[1],lab2);   Kode for S  Rek. kall
      sprintf(codestr,"%s %s","ujp",lab1);
      emitCode(codestr);
      sprintf(codestr,"%s %s","lab",lab2);
      emitCode(codestr);
      break;
    case BreakKind:
      sprintf(codestr,"%s %s","ujp",label);
      emitCode(codestr);
      break;
    case OtherKind:
      emitCode("Other");
      break;
    default:
      emitCode("Error");
      break;
  }
```

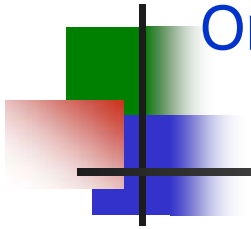
Rekursiv prosedyre for P-kodegenerering for setninger, penere utgave.



Merk: Stakken antas å være tom før og etter kodegen. for setning, men at stakken øker med én i løpet av kodegen. for uttrykk.

En ”break” i kildeprogr. skal bli et hopp til denne labelen. Den vi angi første instruksjon etter nærmest omsluttende while-setning.

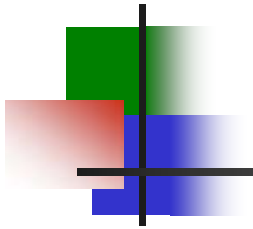
```
void genCode(TreeNode t, String label){
  String lab1, lab2;
  if t != null{  // For et tomt tre, ikke gjør noe
    switch t.kind {
      case ExprKind {  // I boka (forrige foil) er det veldig forenklet.
                       // Kan behandles slik uttrykk er behandlet tidligere
      }
      case IfKind {      // If-setning
        genCode(t.child[0], label); // Lag kode for det boolske uttrykket.  Brukers egentlig label her?
        lab1= genLabel();
        emit2(”fjp”, lab1);  // Hopp til mulig else-gren (eller til slutten om ikke else-gren)
        genCode(t.child[1], label); // kode for then-del, gå helt ut om break opptrer
        if t.child[2] != null {     // Test på om det er else-gren?
          lab2 = genLabel();
          emit2(”ujp”, lab2);   //  Hopp over else-grenen
        }
        emit2(”label”, lab1);     // Start på else-grenen, eller slutt på if- setningen
        if t.child[2] != null {     // En gang til: test om det er else-gren? (litt plundrete programmering)
          genCode(t.child[2], label); // Kode for else-gren, gå helt ut om break opptrer
          emit2(”lab”, lab2);   // Hopp over else-gren går hit
       } }
       case WhileKind {  /* mye som over, men OBS ved indre ”break”. Se boka og forrige foil */ }
       case BreakKind { emit2(”ujp”, label); }  // Hopp helt ut av koden som dette genCode-kallet lager
       ...                                      //  (og helt ut av nærmest omsluttende while-setning)
} } }
```



Om lur kodegen. av boolske uttrykk i if- og while-setninger

ER pensum, og det står litt om dette i kap 8.4.3.
Men det meste av stoffet finnes bare på disse foilene.

- Vi har så langt tenkt at vi genererer kode for boolske uttrykk omtrent som for aritmetiske uttrykk. Men ...
  - Boolske uttrykk kan imidlertid bare ha to verdier (*true* og *false*), og dette gjør at bergningen kan forenkles.
  - Dette gjelder spesielt om uttrykket forekommer som betingelse i en while- eller if-setning
  - Mange språk foreskriver at *and* og *or* i boolske uttrykk skal beregnes slik at om man etter første operanden vet svaret på hele opersjonen, så skal man *ikke* beregne den andre operanden.
    - Dette er omtalt i boka, kap. 8.4.3, og kalles der “short circuit”-beregning
    - Vi kaller det på norsk “kortslutning” av boolske uttrykk.
    - Vi vil i fortsettelsen anta at det er slik



Eksempel på TA-kode for kortsluttet beregning

Vi antar altså at språket foreskriver at *and* og *or* i boolske uttrykk skal beregnes slik at om man etter første operanden vet svaret på hele opersjonen, så skal man *ikke* beregne den andre operanden

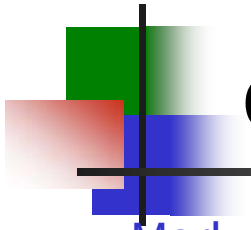
Vi skal altså kortslutte i følgende tilfeller:

- For *or* -operasjon om første operand blir *true* (hele svaret er da *true*)
- For *and* -operasjon om første operand er *false* (hele svaret blir da *false*)

Oversett (pr hånd) følgende setning:

if a<b || (c>d && e>=f) then x=8 else y=5 endif

```
t1 = a < b
if_true t1 goto 1    // Vi vet at uttrykket er sant
t2 = c > d
if_false t2 goto 2   //  Vi vet at uttrykket er galt
t3 = e >= f
if_false t3 goto 2   // Vi vet at uttrykket er galt, ellers er det sant og vi fortsetter
label 1
x = 8
goto 3
label 2
y = 5
label 3
```



Om å lage kodegenerator for kortslutning etc.

- Merk at dette **blir *litt* komplisert** om man skal generere P-kode (slik vi tidligere gjorde), fordi man da må tenke på at stakkdybden etc. må stemme overens ved hoppet og ved lablen. Vi går derfor over til å lage TA-kode.

- Hovedsaken med den koden som er angitt på forrige foil er altså at man aldri *beregner* (den logiske verdien) på hele det boolske uttrykket, og at man, så fort man vet om hele uttrykket (eller et subuttrykk) blir true eller false, så hopper man *direkte* dit dette tilfellet skal behandles videre.

- Hvordan slik kode kan genereres vil avhenge veldig om vi bare skal ha symbolske labeler som vi kan velge fritt, eller om hoppene skal gå til faktiske fysiske adresser, f.eks. til der else-grenen starter (som ikke er kjent når det boolske uttrykket behandles).

  - Det første tilfellet behandles på de neste foilene

  - Det siste tilfellet er mer fiklete, men ikke prinsippielt annerledes. Det er ikke med som pensum (men er litt omtalt i kap. 8.4.2).

Vi skal lage *tekstlig TA-kode*, og lar oss inspirere av (den gamle) foilen under. Men merk at denne altså lager P-kode, men på de neste foilene lages TA-kode

Til denne labelen skal en ”break” i kildeprogrammet gå

```
void genCode(TreeNode t, String label){
   String lab1, lab2;
   if t != null{  // Er vi falt ut av treet?
      switch t.kind {
         case ExprKind {  // I boka er dette veldig forenklet, ved at alt er konstanter true  og false
                          // Men kan behandles slik uttrykk er behandlet tidligere
         }
         case IfKind {      // If-setning
            genCode(t.child[0], label); // Lag kode for det boolske uttrykket
            lab1= genLabel();
            emit2(”fjp”, lab1);  // Hopp til mulig else-gren, eller til slutten av for-setning
            genCode(t.child[1], label); // kode for then-del, gå helt ut om break opptrer (inne i uttrykk??)
            if t.child[2] != null {     // Test på om det er else-gren?
               lab2 = genLabel();
               emit2(”ujp”, lab2);   //  Hopp over else-grenen
            }
            emit2(”label”, lab1);     // Start på else-grenen, eller slutt på if- setningen
            if t.child[2] != null {     // En gang til: test om det er else-gren? (litt plundrete programmering)
               genCode(t.child[2], label); // Kode for else-gren, gå helt ut om break opptrer
               emit2(”lab”, lab2);   // Hopp over else-gren går hit
          } }
          case WhileKind {  /* mye som over, men OBS ved indre ”break”. Se boka */ }
          case BreakKind { emit2(”ujp”, label); }  // Hopp helt ut av koden dette genCode-kallet lager
          ...                                      //  (og helt ut av nærmest omsluttende while-setning)
} } }
```

Kodegenerering for If-setning til TA-kode

- Vi lager behandlingen av if-setning ved å justere på den fra forrige foilen (som laget P-kode)
- Det å gå over fra P-kode til TA-kode blir knapt synlig her.
- Vi antar nå at det ikke er noen break-setning
- Men vi må likevel gi med parametere til metoden som behandler det boolske uttrykket:
  - Disse parametrene er to strenger, som er navn på "planlagte" labler. Disse vil angi hvor det skal hoppes når det blir klart om svaret er hhv. true eller false.

Vi bryr oss ikke med retur-verdien (temp-navn), siden den *alltid vil hoppe* ut

```
.....
case IfKind {
        String labT = genLabel(); String labF = genLabel();  // Skal hoppes til om, betingelse er True/False

        genBoolCode(t.child[0], labT, labF); // Lag kode for betingelsen. Vil alltid hoppe til labT eller labF
        emit2("lab", labT);      // True-hopp fra betingelsen skal gå hit
        genCode(t.child[1]);   // kode for then-gren (nå uten label-parameter for break-setning)

        String labx = genLabel();      // Skal angi slutten av en eventuell else-gren.
        if t.child[2] != null {        // Test på om det er noen else-gren?
            emit2("ujp", labx);        // I så fall, hopp over else-grenen
        }

        emit2("label", labF);      // False-hopp fra betingelsen skal gå hit
        if t.child[2] != null {    // En gang til: test om det er else-gren? (litt plundrete programmering)
            genCode(t.child[2]); // Kode for else-gren
            emit2("label", labx);     // Hopp forbi else-grenen går hit
}       }
.....
```

Genere TA-kode for boolske uttrykk

Men heller ikke *denne* vil lage helt god kode!
Hvorfor?? (se helt nederst)

Vi bryr oss ikke med retur-navnet, siden de alltid vil hoppe ut

```
void genBoolCode(String labT, labF) {
        ...
        case ”||”: {
            String labx = genLabel();
            left.genBoolCode(labT, labx);
            emit2(”label”, labx);
            right.genBoolCode(labT, labF);
        }
        case ”&&”: {
            String labx = genLabel();
            left.genBoolCode(labx, labF);  // som over
            emit2(”label”, labx);
            right.genBoolCode(labT, labF);  // som over
        }
        case ”not”: { // Har bare ”left”-subtre
            left.genBoolCode(labF, labT);  // Ingen kode lages!!!
        }
        case ”<”: {
            String temp1, temp2, temp3; // temp3 skal holde den boolsk verdi for relasjonen
            temp1 = left.genIntCode(); temp2 = right.genIntCode();      temp3 = genLabel();
            emit4(temp3, temp1, «lt», temp2); // temp3 får (det boolske) svaret på relasjonen
            emit3(«jmp-false», temp3, labF);
            emit2(«ujp», labT);  // Denne er unødvendig dersom det som følger etter er labT
                                 // Dette kan vi oppdage med en ekstra parameter som angir labelen bak
}       }                        // den konstrusksjonen man kaller kodegenererings-metoden for.
```

For ”||”: