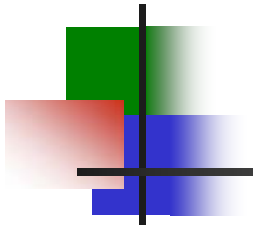
INF5110 – 12. og 13. april, 2011
Kap. 8 – kodegenerering

Endelig utgave 13/4

Stein Krogdahl,
Ifi UiO



Kodegenerering

Denne uken, fra kapittel 8.
Dette er nok til Oblig 2, og er stort sett uavhengig av maskin-detaljer, registre etc.

8.1 Bruk av mellomkode
8.2 Basale teknikker for kode- generering
8.3 Kode for referanser til datastrukturer (ikke alt)
8.4 Kode for genereering for kontroll-setninger og logiske uttrykk
Kanskje også noe fra: 8.5, 8.9 og 8.10

Fra 4. mai:

- Oppkopiert stoff: Om lur bruk av registre til å holde data som snart skal brukes om igjen
- Noe om generiske mekanismer i programmeringsspråk
- Litt mer om Javas byte-kode

NB: Detaljert pensumliste kommer!

Oversikt (se figurer på neste foiler)

- 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 *veldig maskinnær* mellomkode).
  - Genereres tradisjonelt på én av 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

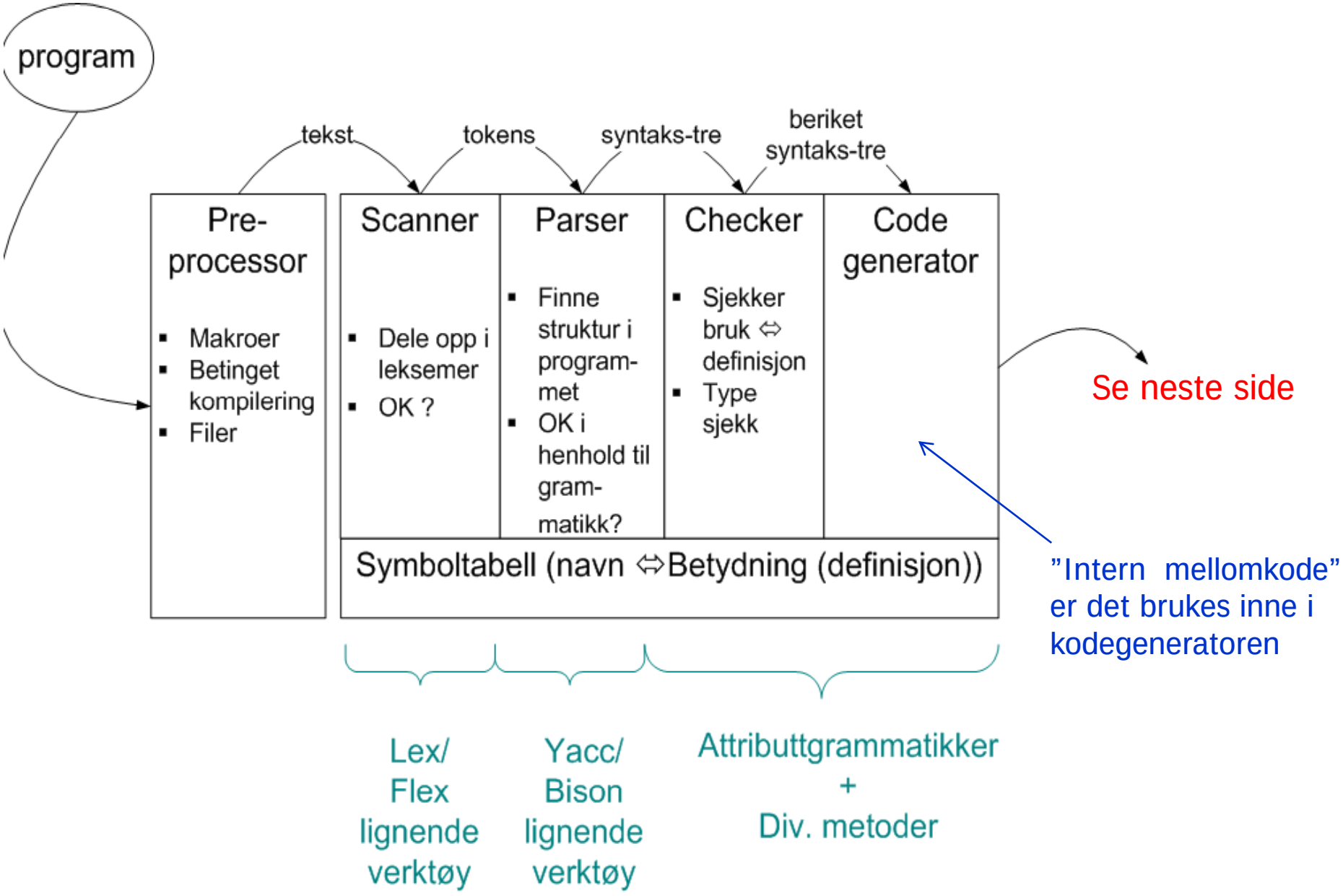
  (3) Direkte som binær maskin-kode (som ev. kan legges rett ned i maskinen)

- Men: Det kan også være greit å overføre programmet til en halvkompilert form, som oftest er uavhengig 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

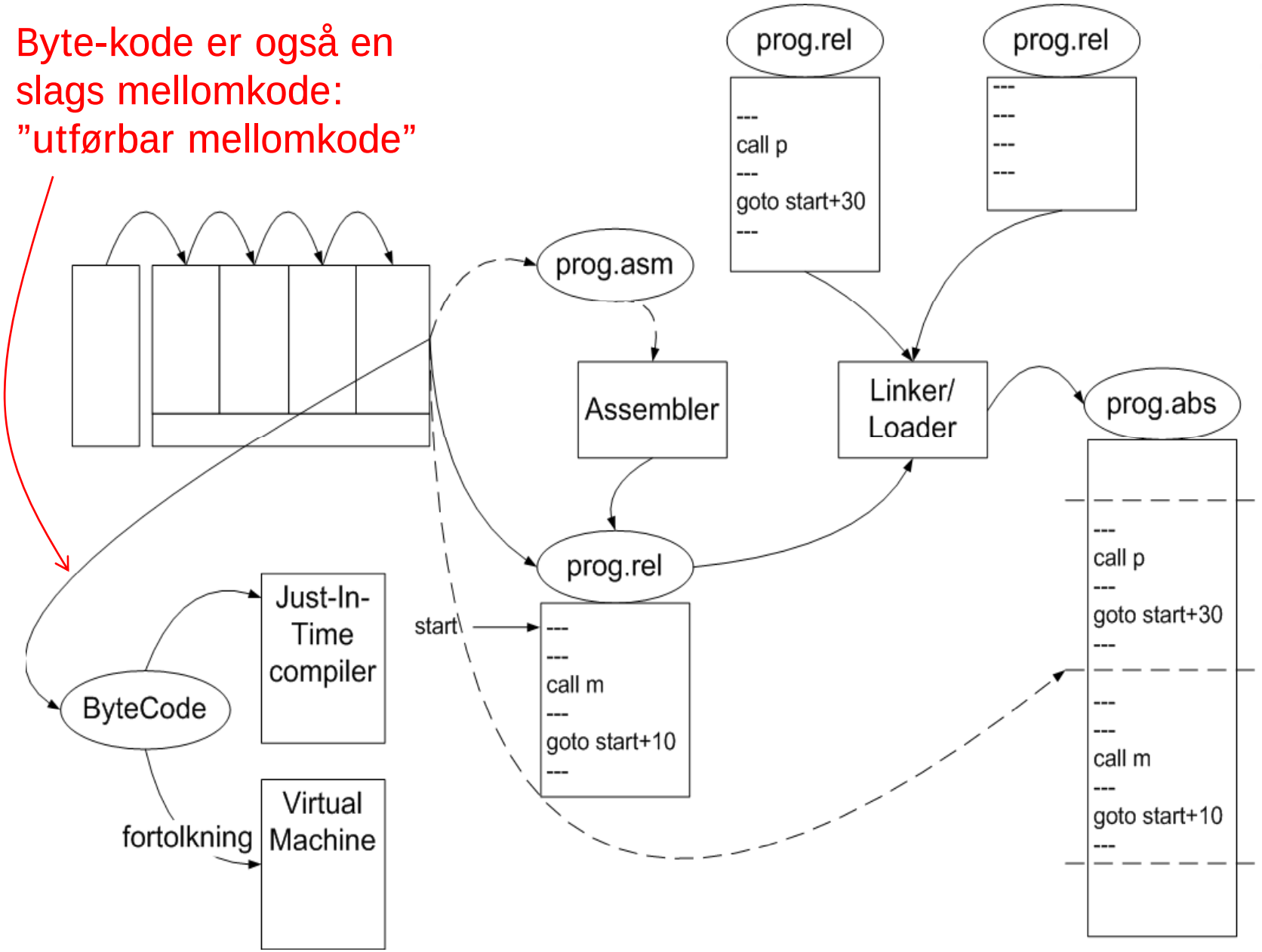
    (2) Utførbar mellomkode (som ofte også blir kompilert videre til maskinkode)
  - Disse formene er nokså like, og vi skal her snakke om begge typer samlet (men i det vi tar med fra kap. 8 snakker boka mest om den *interne* typen)

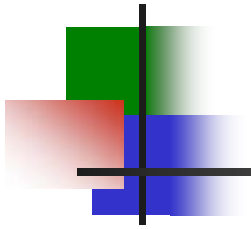
Fra første forelesning: Anatomien til en kompilator



Anatomien til en kompilator - II

Byte-kode er også en slags mellomkode: ”utførbar mellomkode”

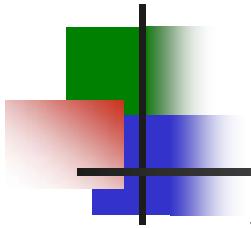




8.1 Bruk av mellomkode

- Vi skal se på to ”stilarter” for slik mellom-kode:
  - Treadresse-kode (TA-kode)
    - Setter navn på mellomresultater (kan tenkes på som registre)
    - Forholdsvis lett å snu om på rekkefølgen av koden (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 fomatene, f.eks.:
  - Er adresser oversatt til binærform, eller er de på tekstlig form?
  - Er det egne operasjoner for f.eks. array-aksess, eller blir slike større operasjoner delt opp i flere enklere ”instruksjoner”?



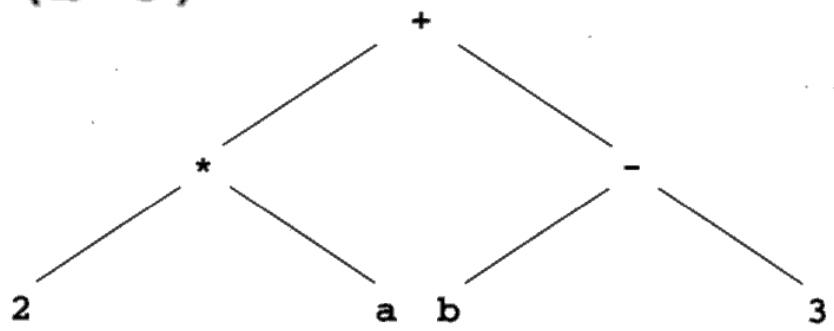
Vi skal se på en del oversettelser:

- 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
    - Denne er ikke så lett å få effektiv
- Man kommer da 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
  - 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)
```



Tre-adresse (TA) kode

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

En alternativ kode-sekvens

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

t_1, t_2, t_3,... er temporære variable

TA grunnform:

x = y *op* **z**

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

Også:

x = op y

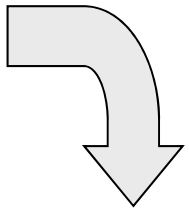
op = not, -, float-to-int. ...

Andre TA-instruksjoner:

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

”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?

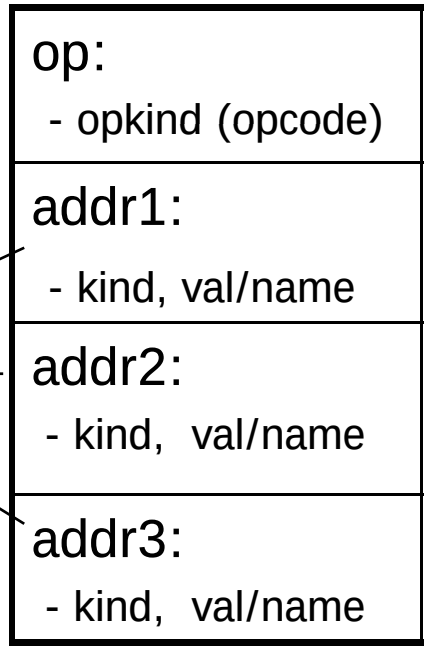
- - kvadrupler, de tre adressene, og operasjonen
- - (tripler, der ”adressen” til instruksjonen er navn på en ny temporær variabel)

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;
```

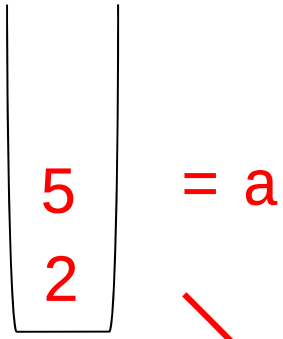
Hver adresse har denne formen



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

”push” koder (= load på stakken):

```
lod   ;  load value
ldc   ;  load constant
lda   ;  load adress
```



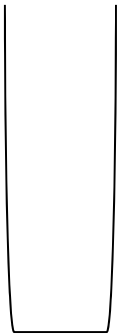

```
2*a+(b-3)
```

```
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

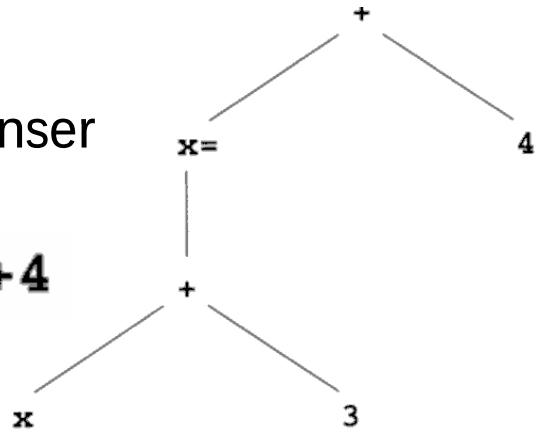
NB: Direkte eksekvering av denne er veldig upraktisk

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

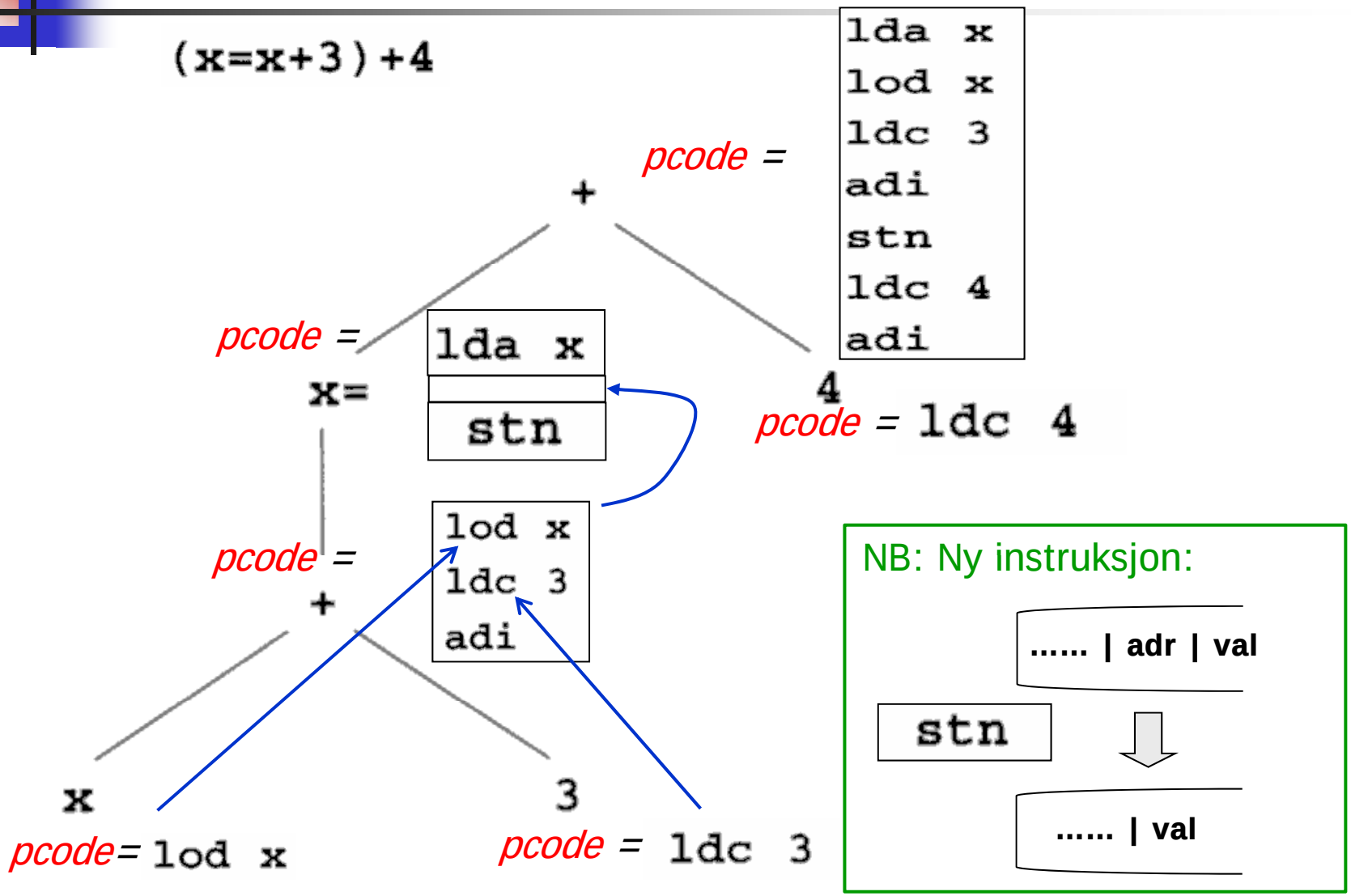
++ betyr: sette sammen instruksjonsekvenser

(x=x+3)+4



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

(Vil altså aldri gjøre det slik i praksis!)

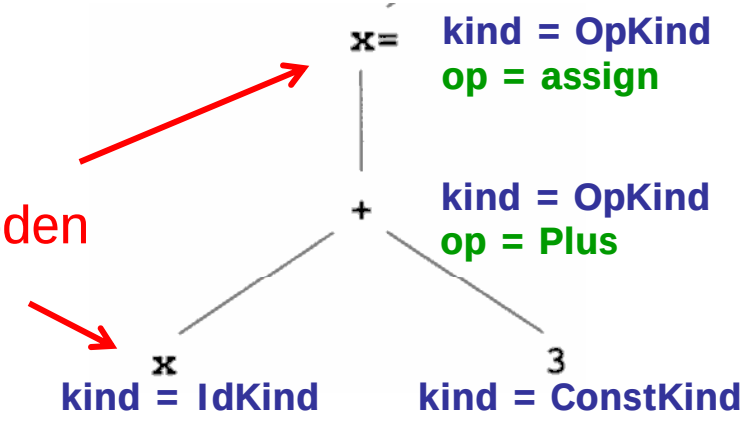


Kodegenerering kan gjøres ved rekursiv gjennomgang av syntakstreet. Forslag til tre-node:

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



Metode-skisse til generelt bruk

ved rekursiv traversering av binære trær

```
procedure genCode ( T: treenode );
begin
   if T is not nil then
      generate code to prepare for code of left child of T ;      ← Prefiks - operasjoner
      genCode(left child of T ) ;                                 ← rekursivt kall
      generate code to prepare for code of right child of T ;     ← Infiks - operasjoner
      genCode(right child of T ) ;                                ← rekursivt kall
      generate code to implement the action of T ;                ← Postfiks - operasjoner
end;
```

Generering av P-kode fra tre-struktur (som i Oblig2)

```
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;
  }
}
}
```

Angivelse av TA-kode ved attr.-grammatikk

`(x=x+3)+4`

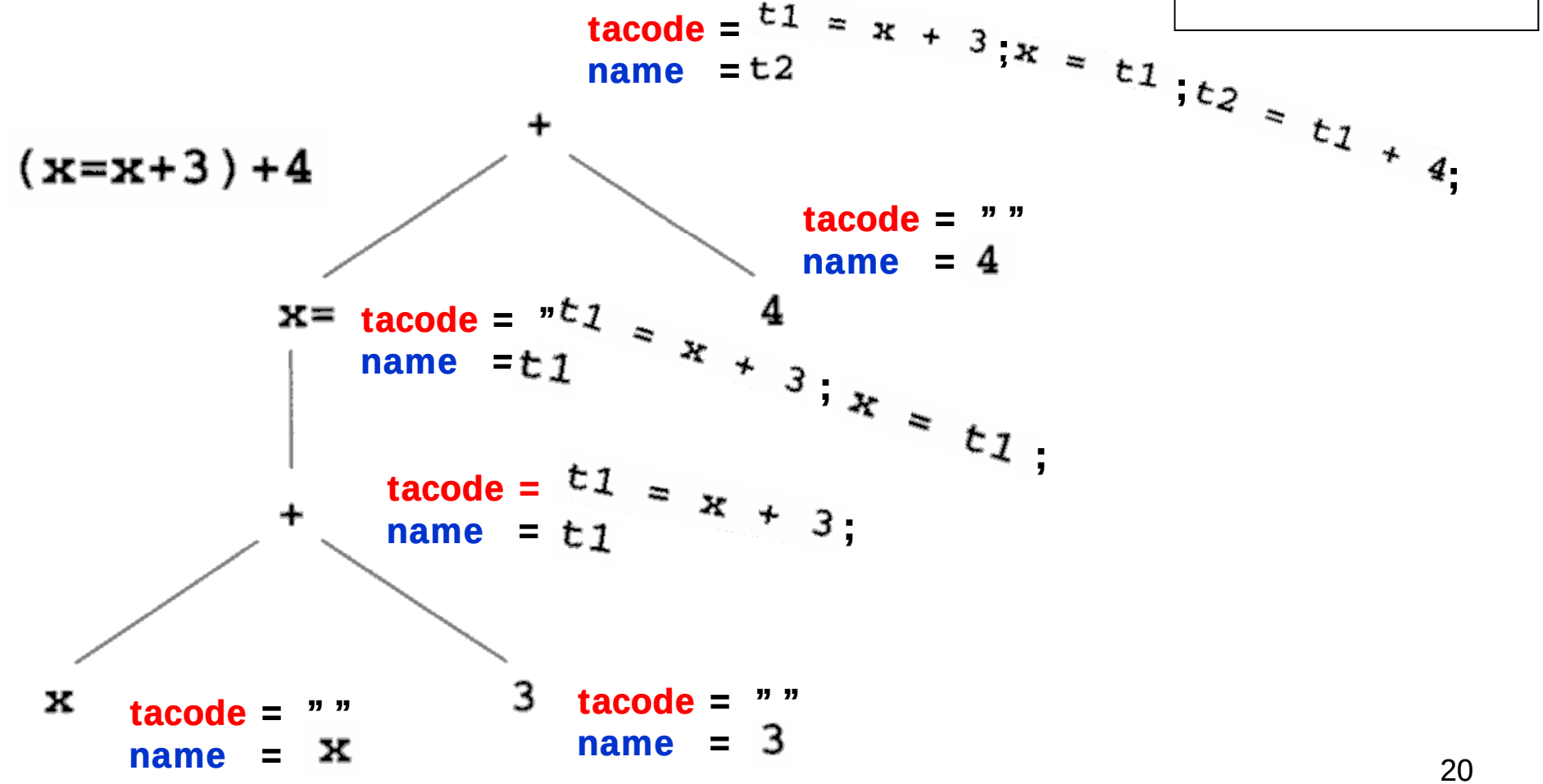
Grammar Rule	Semantic Rules
$exp_1 \rightarrow$ **id** = exp_2	exp_1 .name = exp_2 .name exp_1 .tacode = exp_2 .tacode ++ **id** .strval ‖ "=" ‖ exp_2 .name
$exp \rightarrow aexp$	exp .name = aexp .name exp .tacode = aexp .tacode
$aexp_1 \rightarrow aexp_2$ + factor	$aexp_1$ .name = newtemp( ) $aexp_1$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



Generering av rent tekstlig TA-kode fra tre-struktur

Hoved-delen av en rekursiv metode.

Metoden eksekverer inne i noden og leverer et navn eller en konstant-streng:

```
switch kind {
  case OpKind:
    switch op {
      case Plus: { tempnavn = nytt temporær-navn;
                   opnavn1 = rek kall for venstre subtre;
                   opnavn2 = rek kall for høyre subtre;
                   emit (”tempnavn = opnavn1 + opnavn2”);
                   return (tempnavn); }
      case Assign: { varnavn = id. for v.s.-variabel (ligger i noden);
                   opnavn = rek kall for venstre subtre;
                   emit (”varnavn = opnavn”);
                   return (varnavn); }
    }
  case ConstKind: { return (konstant-streng); }  // ”Emitter” ingenting!
  case IdKind:      { return (identifikator); }       // ”Emitter” ingenting!
}
```

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

Typisk slik det gjøres i JIT-kompilering av bytekode

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

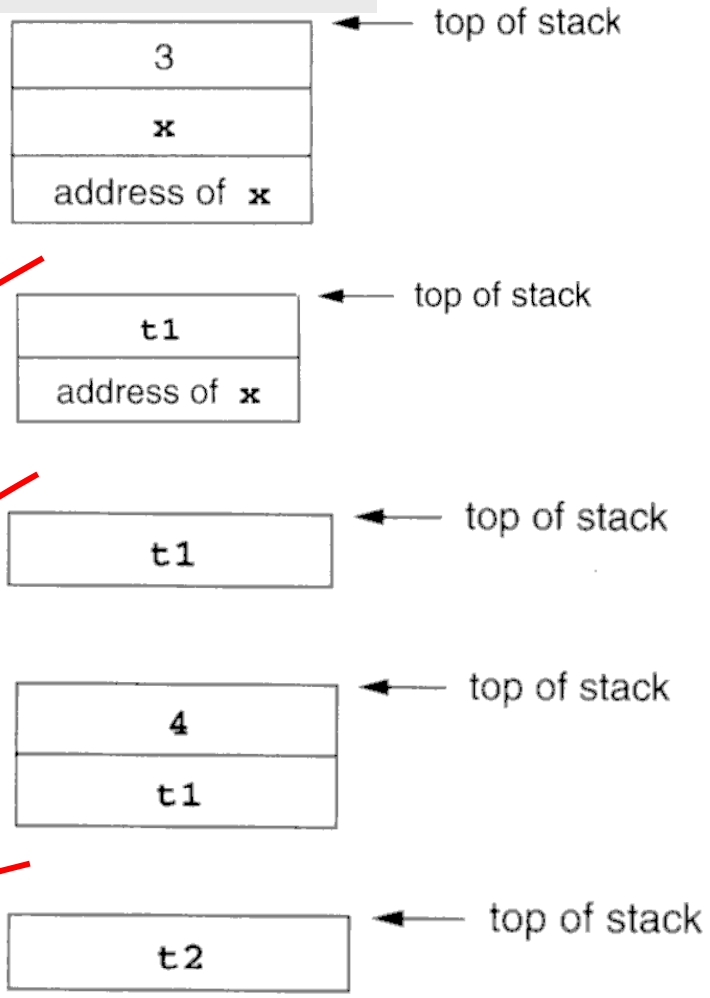
P-kode:

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

Ønskemål:

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

Stakkens stadier:



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
```

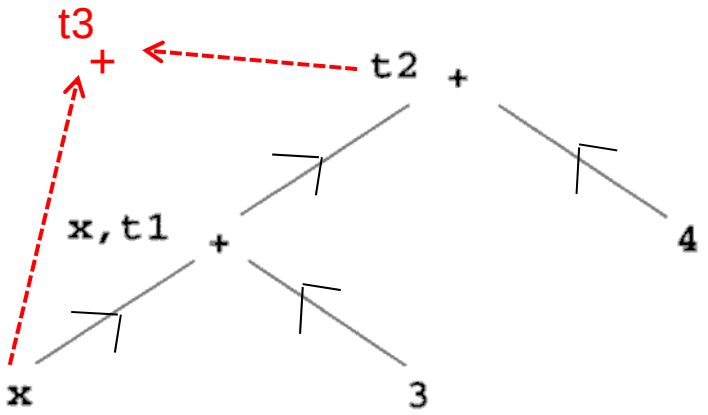
Ny versjon: 13 instr.
Gml. ver. : 7 instr

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

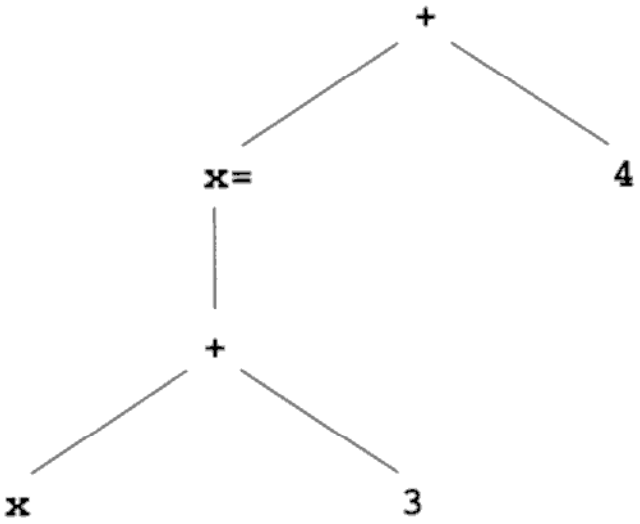
Prøver å lage bedre kode

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

Tegner opp ”data-flyt”-grafen / treet



Må gjøre forskjell på temporære og program-variable.
Kan da se det som:



Generelt: Blir en rettet graf uten løkker (DAG)
Legg til instrusjonen: t3 = x + t2
Derved blir det generelt verre
Men som input til register-allokering er TA-kode OK

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

En mulig løsn.:

sto t1

lod t1
adi

Kap. 8.3 – Detaljert aksess av datastruktur

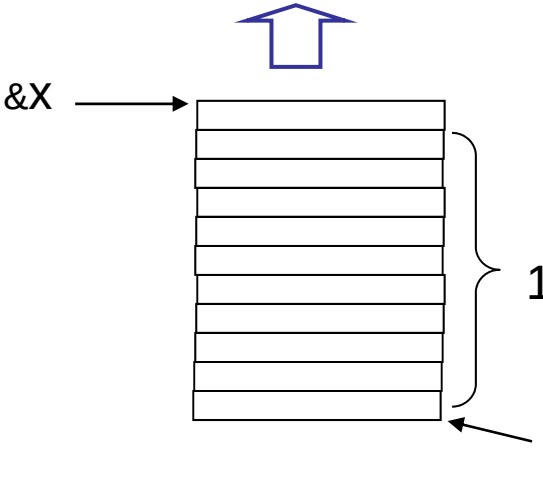
Trenger da flere instruksjoner til adresse-beregning

TA-kode: To nye måter å adressere på

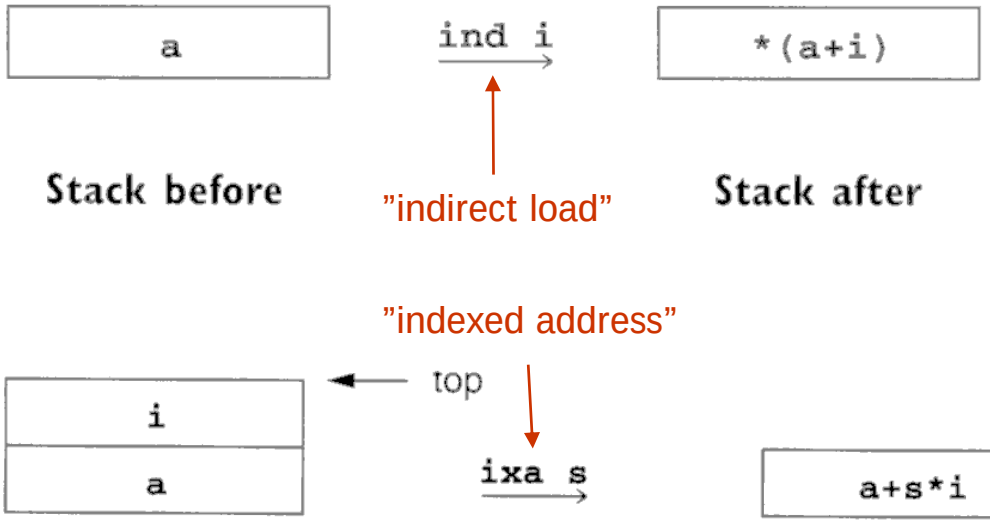
& x Adressen til x (ikke for temporære)

***t** Indirekte gjennom t

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



P-kode: To nye instruksjoner

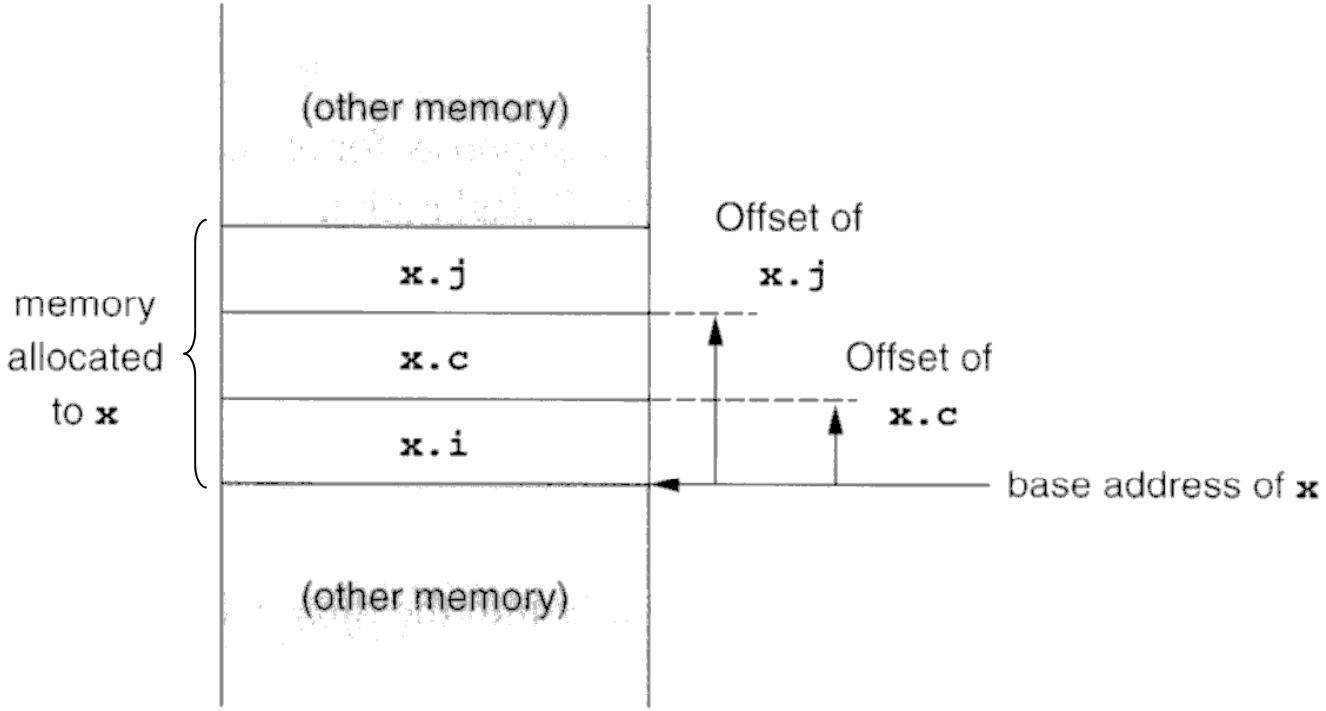


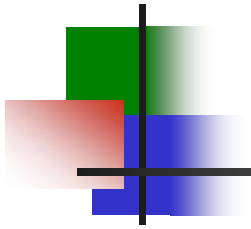

```
lda x
ldc 10
ixa 1
ldc 2
sto
```

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;
```



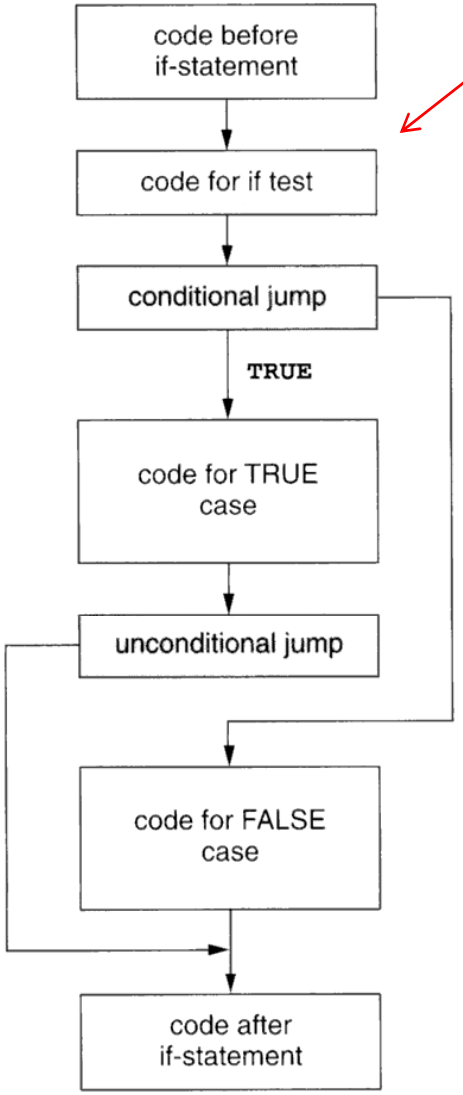


Litt generelt til kap. 8.3

- I boka lages det nokså ”lavnivå” TA-kode og P-kode
  - Ulempe: Man kan da ikke lenger se hva slags språk-konstruksjoner den kommer fra. Vanskligere å optimalisere.
- Det er ikke opplagt at bokas variant er det fornuftigste. Alternativ f.eks:
  - Beholde en ikke-lokal eller ikke-global variabel på formen:
    X: (rel.niv.=2, reladr=3)
  - Istedenfor å oversette til formen:
    fp.al.al.(reladr=3) i TA-kode eller P-kode
- Kan kanskje like gjerne se oversettelse til lav-nivå TA-kode eller P-kode som eksempel på oversettelse direkte til maskin-kode
  - Bortsett fra at vi her slipper register-allokerings-problemet

8.4 : If/while

Kan vi gjøre noe lurere her?
Kommer som oppgave.



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

while-stmt → **while** (*exp*) *stmt*

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

Skisse av TA-kode

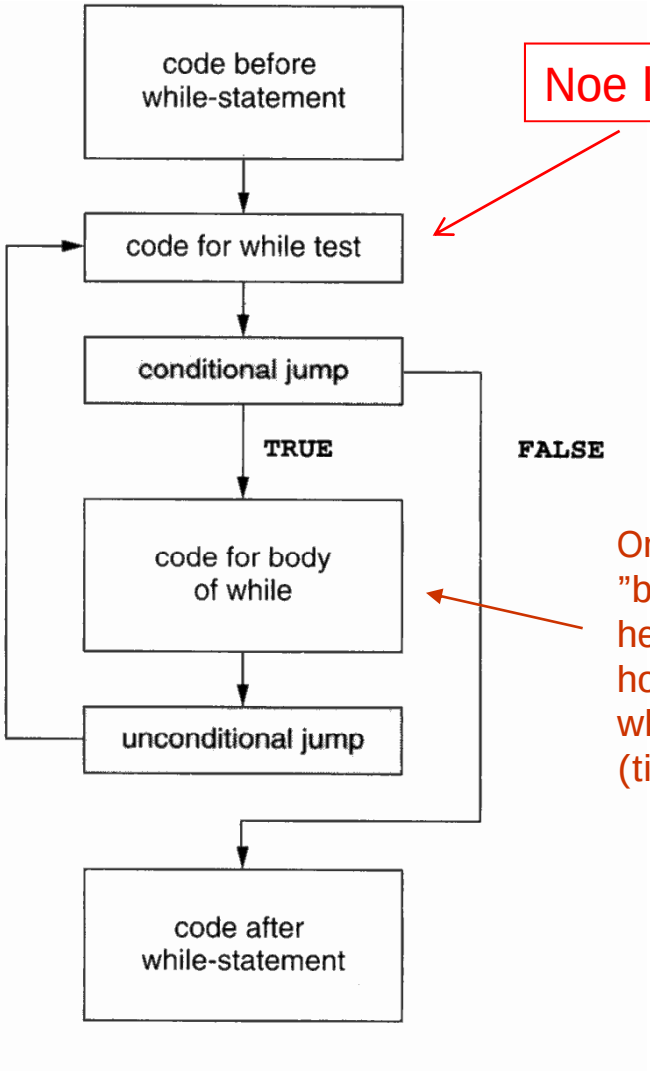
```
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
goto L2
label L1
<code for S2>
label L2
```

Skisse av P-kode

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

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)) ...
```

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

```
(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 hopps 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**

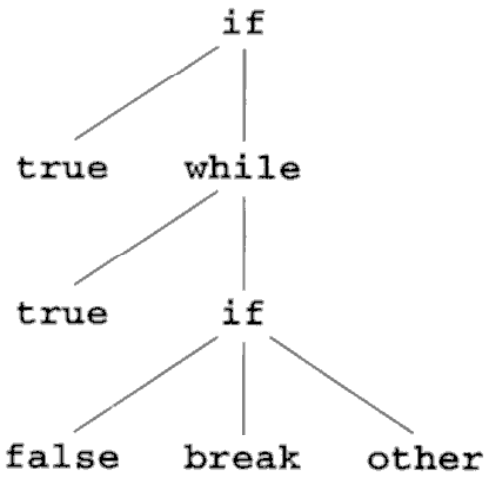
Tre-node:

```
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

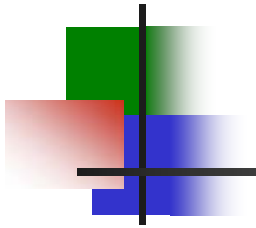
```
void genCode( SyntaxTree t, char * label)
{ char codestr[CODESIZE];
  char * lab1, * lab2;
  if (t != NULL) switch (t->kind)
  { case ExpKind:
      if (t->val==0) emitCode("ldc false")
      else emitCode("ldc true");
      break;
    case IfKind:
      genCode(t->child[0],label);   Rek. kall
 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

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{  // Er vi falt ut av treet?
    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 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 og forrige foil */ }
      case BreakKind { emit2(”ujp”, label); }  // Hopp helt ut av koden dette genCode-kallet lager
      ...                                      //  (og helt ut av nærmest omsluttende while-setning)
} } }
```



Oppgave med svar

(Senere oppgave: Hvordan *generere* slik kode?)

a) Oversett (pr hånd) følgende setning til TA-kode:

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

Oversett den til kode der alle hopp blir så direkte som over hodet mulig (uten å tenke på algoritmen for å gjøre det).

SVAR:

```
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
```