Kap. 8 del 1 – kodegenerering INF5110 – Vår2007

Stein Krogdahl,
Ifi UiO

Forelesninger framover:
Tirsdag 8. mai: Vanlig forelesning
Torsdag 10. mai: *Ikke* forelesning
Tirsdag 15. mai: Vanlig forelesning (siste?)

Pensumoversikt - kodegenerering

8.1 Bruk av mellomkode

8.2 Basale teknikker for kode-generering

8.3 Kode for referanser til datastrukturer (ikke 8.3.2)

8.4 Kode for generereing for kontroll-setninger og logiske uttrykk

(8.5 Kode-generering for prosedyrer og kall)

---------(resten ikke pensum) ------

8.6 Kode produsert av to kommersielle kompilatorer

8.7 TM: En enkel maskin

8.8 Kode-gen for Tiny på TM

8.9 og 8.10 Optimalisering

+**Pensum:**

En del fra utdelt kap 9 fra Aho, Sethi og Ullmann ' s kompilatorbok ("Drage-boka") :

9.2,9.4, 9.5 og 9.6

NB: De aktuelle sidene blir kopiert opp av kursledelsen og blir delt ut neste gang

Hvordan er instruksjonene i en virkelig CPU?

- Ofte: Et antall Registere (8-128) hvor add, mult, sub, shift … går mellom disse
- Men på noen maskiner også disse opersjoner til/fra memory
- Alltid load og store fra register til/fra memory
- Base- og indeks-registre (spesielle registre, eller alle kan være det)
- En instruksjon brytes ned i en rad mikro-instruksjoner
- Viktig:
  - Pipe-line (opp til 22 mikro-instr. er under utførelse samtidig!)
  - "Spekulativ" utføring:
    - av neste instruksjoner, men må restarte om "denne" instr. forandrer dere input-data
    - Ved betingede hopp: Utfører 'begge' grener, men gir opp den ene når valget er klart
- Finnes få rene stakk-baserte maskiner.
  - Men mange har pop/push-aktige instruksjoner, der et gitt register automatisk brukes som topp-av-stakk-peker.

Intel – Utviklet fra 8bit-16bit-32bit. Nå også 64 bit.

- Variabelt format – 722-sider manual – noen hundre instr.

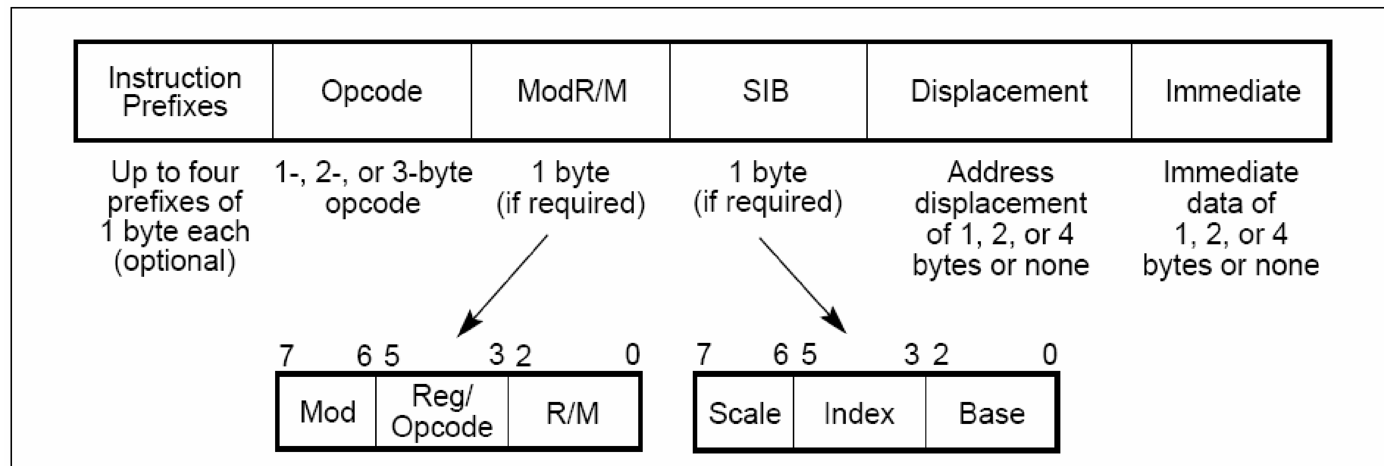


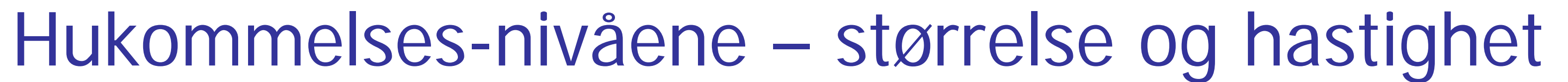
Figure 2-1. IA-32 Instruction Format

5.1.2 Binary Arithmetic Instructions

The binary arithmetic instructions perform basic binary integer computations on byte, word, and doubleword integers located in memory and/or the general purpose registers.

ADD	Integer add
ADC	Add with carry
SUB	Subtract

Hukommelses-nivåene – størrelse og hastighet



Lagertype	Størrelse	Rel. hastighet
Register	8-128	1 (nå ca. 1 ns)
cache L1 data og instr.	12-512 kb	1-2
cache L2	0.5-2Mb	5-10
cache L3 (Itanium)	noen Mb	?
hoved Mem	0.5-2 Gb	100
Disk	80-800 Gb	10 000 000 (ca. 10 ms)

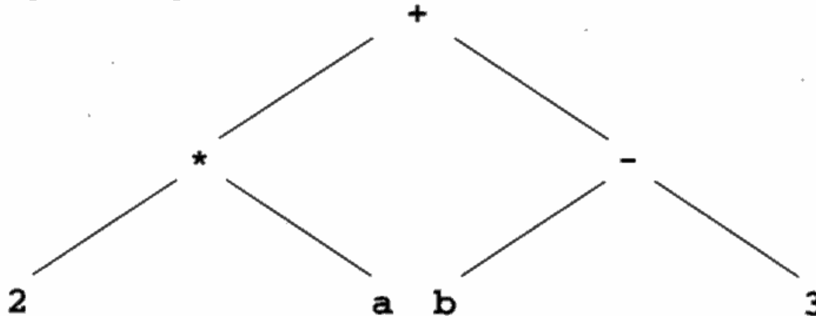
8.1 Bruk av mellomkode

- Man kan godt generere maskinkode direkte fra syntakstreet
- Men: Det kan være greit å overføre programmet til en lineær form: ″mellom-kode″
- Vi skal se på to former:
  - 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″)
    - var opprinnelig beregnet på interpretering
    - Mellomresultatene på en stakk (operasjonene komme postfiks)
- Mange valg, f.eks.:
  - Bevarer vi symboltabellene?
  - Er det operasjoner for array-aksess (eller blir de løst opp i flere, enklere operasjoner?)

Vi skal se på en del oversettelser:

- Vi skal først se på følgende:
  - Generering av TA-kode ut fra tre-strukturen fra sem. analyse
  - Generering av P-kode ut fra tre-strukturen fra sem. analyse
  - Generering av TA-kode fra P-kode
  - Generering av P-kode fra TA-kode
    - Denne er ikke så lett å få effektiv
- Pensum fra annen bok (Ahu, Sethi og Ullmann):
  - Dette deles ut ferdig kopiert opp neste gang
  - Har definisjon av en litt forenklet register-maskin
  - Oversettelse fra TA-kode til denne maskinkoden, med noen typiske problemstillinger.

Tre-adresse (TA)-kode - eksempel



TA grunnform

x = y *op* z

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

Også:

x = op y

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

Tre-adresse (TA) kode

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

En alternativ kode

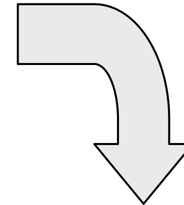
```
t1 = b - 3
t2 = 2 * a
t3 = t2 + t1
```

t_1 t_2, $t_{3,...}$ er temporære variable.

Andre TA-koder:
x = y
if_false x goto L
label L
read x
write x
...

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
```

Spørsmål:

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

-Hvordan er variable representert?

- -ved navn
- -peker til deklarasjon i symb.tabell
- -ved maskinadresse

-Hvordan er hver instruksjon lagret?

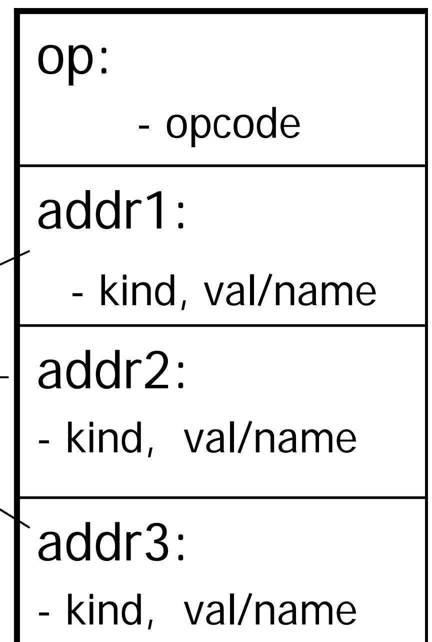
- -kvadrupler
- -tripler der også ”adressen” er navn på en temporær

En mulig datastruktur for å lagre en treadresse-struktur

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) - del I

koden utføres 'normalt (etter hverandre + jump, **men** beregninger på stakk)

"push" koder (= load på stakken):

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

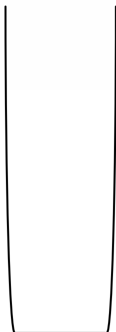
```
2*a+(b-3)
```

```
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")

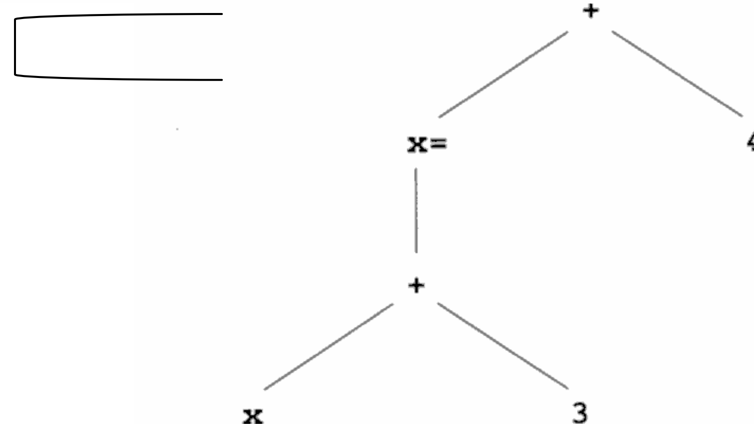
```
1 read x; { input an integer }
2 if 0 < x then { don't compute if x <= 0 }
3   fact := 1;
4   repeat
5     fact := fact * x;
6     x := x - 1
7   until x = 0;
8   write fact { output factorial of x }
9 end
```

```
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
```

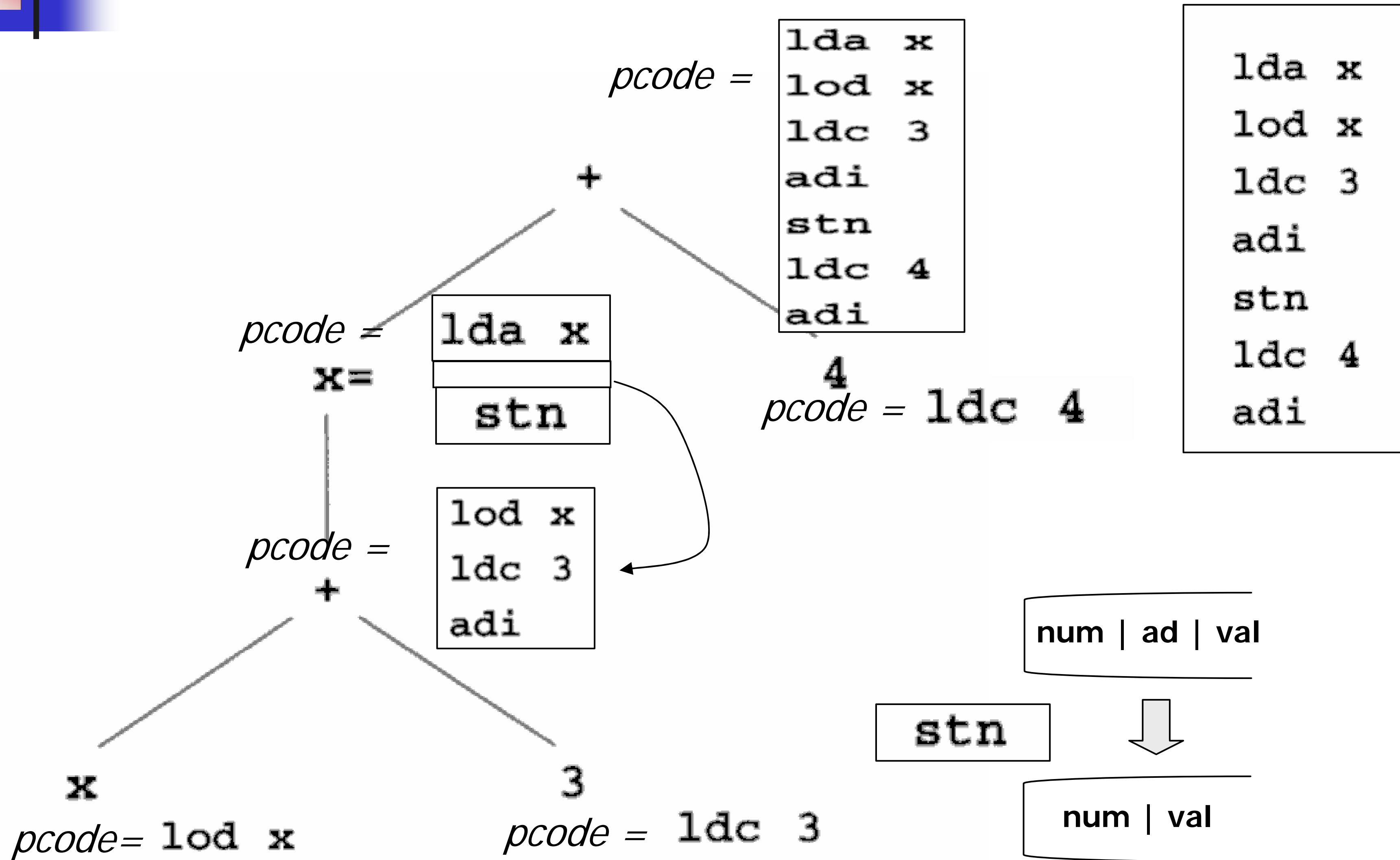
Angivelse av P-kode ved attr.-grammatikk

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

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



Generering av P-kode etter (x=x+3)+4 n.



Angivelse av TA-kode ved attr.-grammatikk

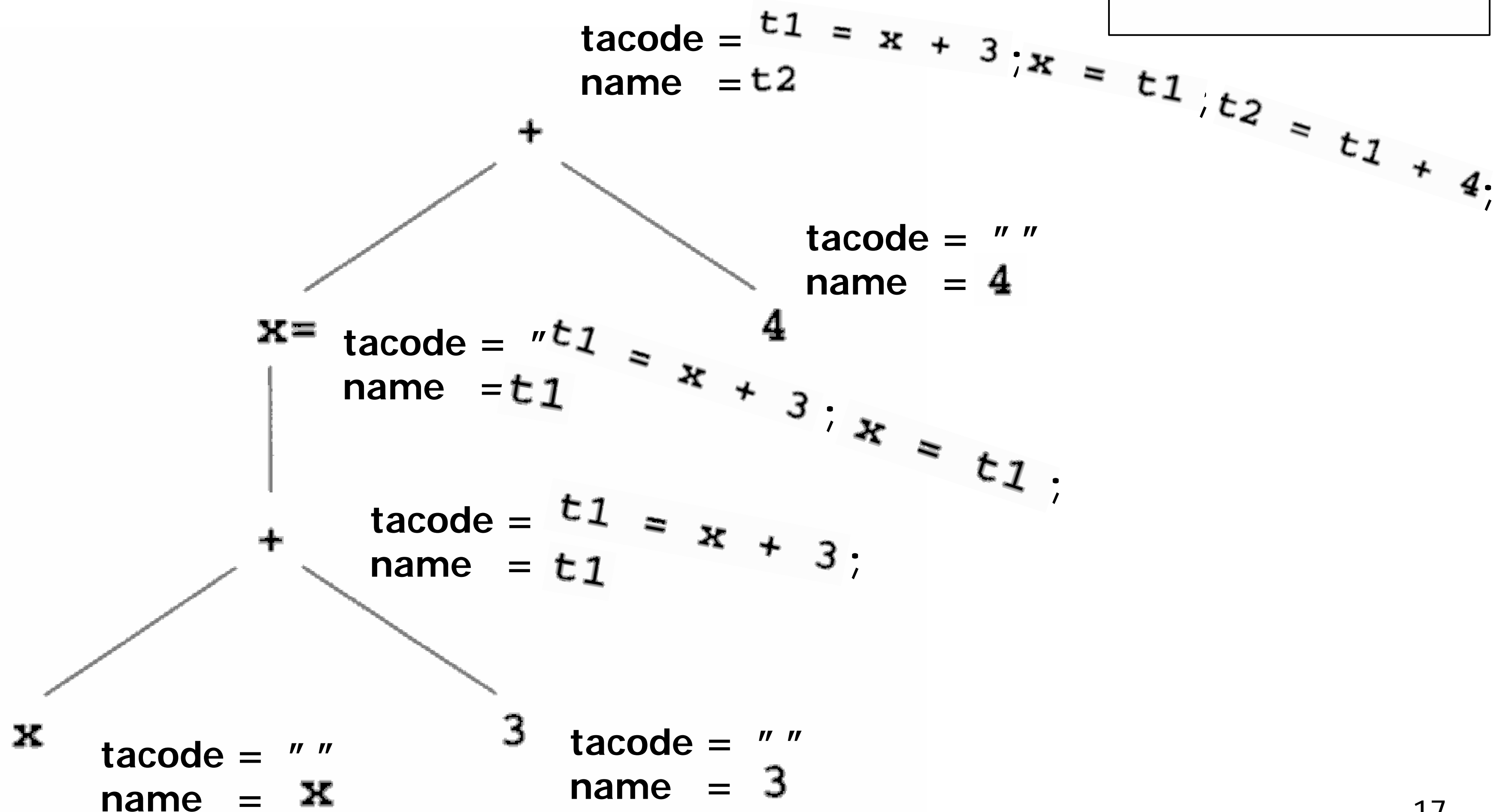
```
(x=x+3)+4
```

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** $.strval$ $factor .tacode$ = **""**

Generering av TA-kode etter attr.-gram.

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

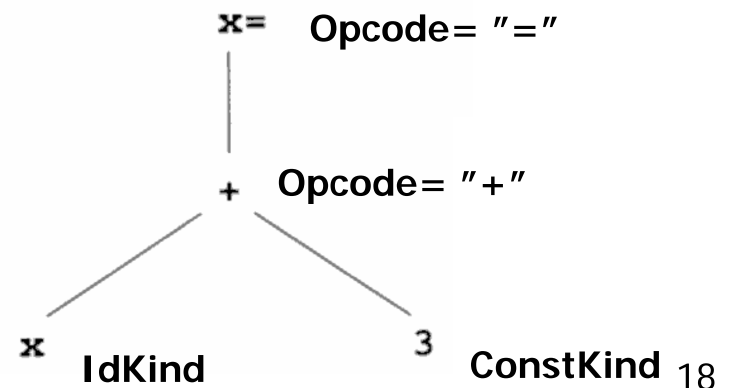
(x=x+3)+4



Kodegenerering kan gjøres ved rekursiv gjennomgang av syntakstreet

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;
```



Kode-skisse til generelt bruk

```
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) ;                    ← rek kall
      generate code to prepare for code of right child of T ;    ← Infiks - operasjoner
      genCode(right child of T) ;                   ← rek kall
      generate code to implement the action of T ;              ← Postfiks - operasjoner
end;
```

Generering av P-kode fra tre-struktur

```
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;
          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;
    }
  }
}
```

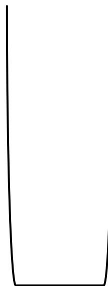
Fra P-kode til TA-kode ("Statisk simulering")

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

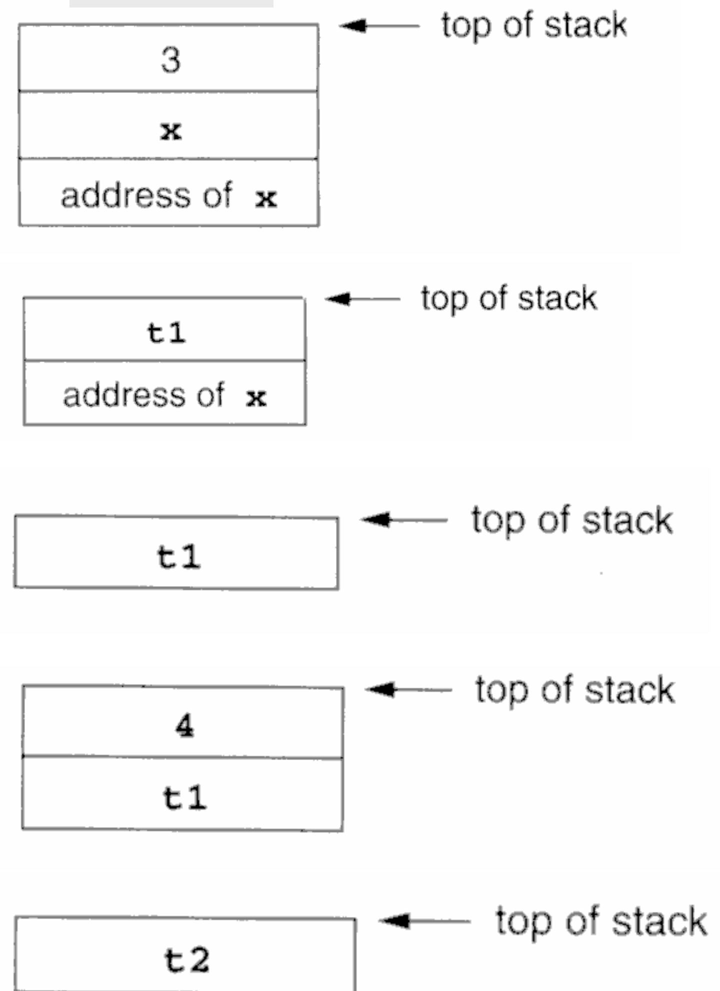
P-kode:

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

Stakk:



Stadier:



Ønskemål:

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

Fra TA-kode til P-kode - med "makro-ekspansjon"

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

Makro:

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

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

t1 = x + 3 →

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

x = t1 →

```
lda x
lod t1
sto
```

t2 = t1 + 4 →

```
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
```

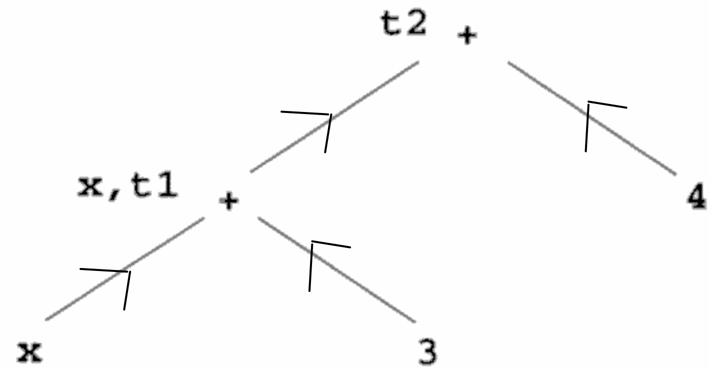
Ny versjon: 13 instr.
Gml. ver. : 7 instr

Fra TA-kode til P-kode - litt lurere, skisse

Prøver å lage bedre kode

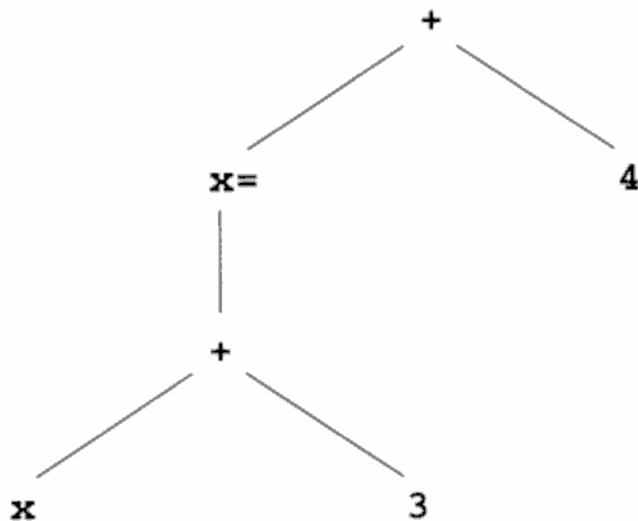
```
t1 = x + 3
x = t1
t2 = t1 + 4
```

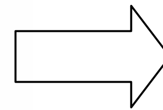
Tegner opp "data-flyt" –grafen / treet



Generelt: En rettet graf uten løkker (DAG)

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





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

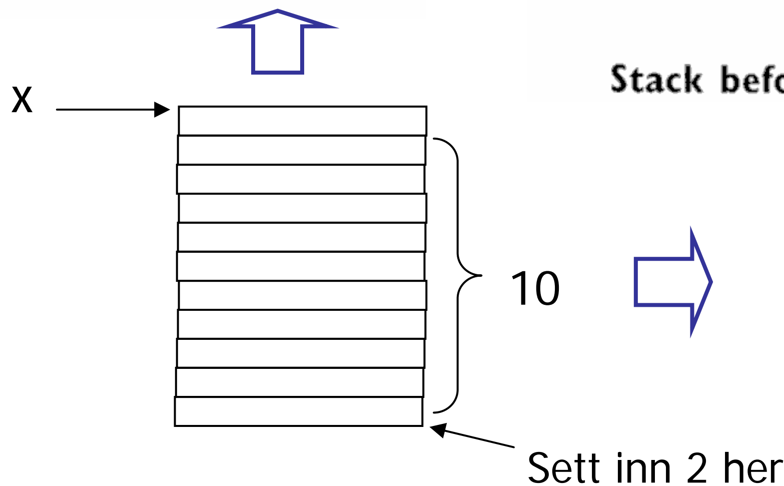
Kap. 8.3 – Aksess av datastruktur, trenger mer fleksibel adresse-beregning

TA-kode:
To nye måter å adressere på

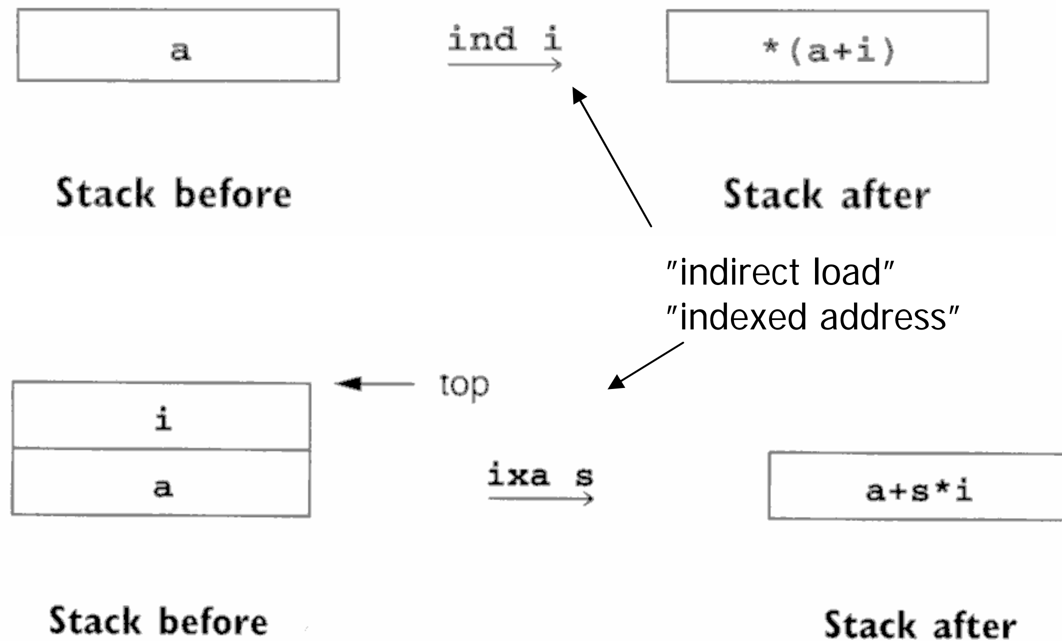
& X Adressen til x
(ikke for temporære)

***t** Indirekte gjennom t

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



P-kode: To nye instruksjoner :




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

Litt generelt til kap. 8.3

- Det lages nokså ″lavnivå″ TA-kode og P-kode
- Man kan ikke lenger se hva slags språk-konstruksjoner den kommer fra

- Det er ikke opplagt at dette er det fornuftigste om mellom-koden skal oversettes videre til maskin-kode, f.eks:
  - Beholde en ikke-lokal eller ikke-global variabel på formen:

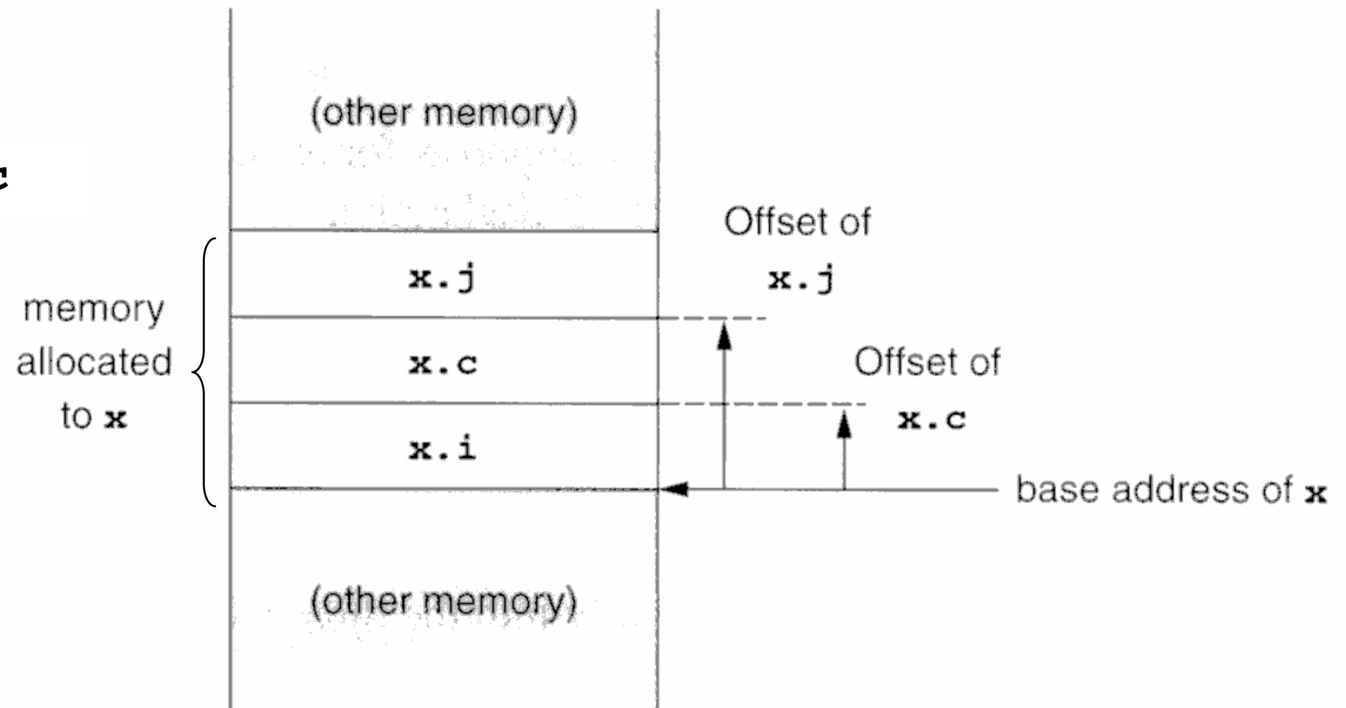
    X: (rel.niv.=2, reladr=3)
  - Istedenfor å oversette til formen:

    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
  - men får da ikke register-allokerings-problemet

TA-kode og P-kode for strukt/record-aksess (Tilsvarende for "fp" til metode-kall)

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



Skal se på

1. En variable av typen Rec ligger statisk allokert (som "x" over)
2. Vi har en <u>peker</u> til en struct/record (f.eks. "fp" til metodekall)

Til TA-kode

Statisk allokert variabel

```
x.j = x.i;
```

```
t1 = &x + field_offset(x,j)
t2 = &x + field_offset(x,i)
*t1 = *t2
```

ville være like logisk å oppgi her struct-typen, altså "rec"

Peker til struct (kan for eksempel være "fp" metode kall)

```
typedef struct treeNode
  { int val;
    struct treeNode * lchild, * rchild;
  } TreeNode;
...
TreeNode *p;
```

Now, consider two typical assignments

```
p->lchild = p;
p = p->rchild;
```

These statements translate into the three-address code

```
t1 = p + field_offset(*p,lchild)
*t1 = p
t2 = p + field_offset(*p,rchild)
p = *t2
```

Eller altså "Tree Node"

Til P-kode

```
int    *X;
```

```
*x = i;
```

translates into the P-code

```
lod x
lod i
sto
```

Her kan C-koden nyttes direkte som TA-kode

and the assignment

```
i = *x;
```

translates into the P-code

```
lda i
lod x
ind 0
sto
```

```
Rec    x;
```

```
x.j = x.i;
```

can be translated into the P-code

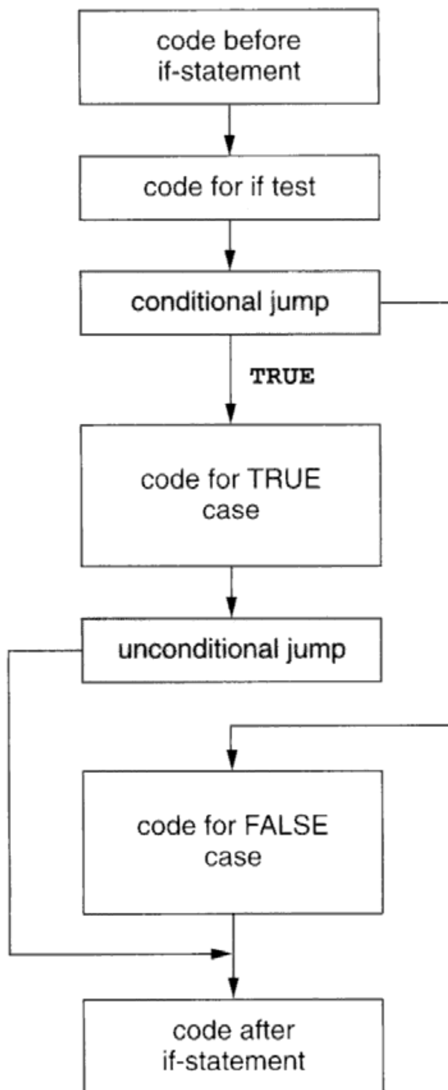
```
lda x
lod field_offset(x,j)
ixa 1
lda x
ind field_offset(x,i)
sto
```

```
TreeNode *p;
```

```
p->lchild = p;
p = p->rchild;
```

```
lod p
lod field_offset(*p,lchild)
ixa 1
lod p
sto
lda p
lod p
ind field_offset(*p,rchild)
sto
```

8.4 : If/while – kap.8.3



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

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

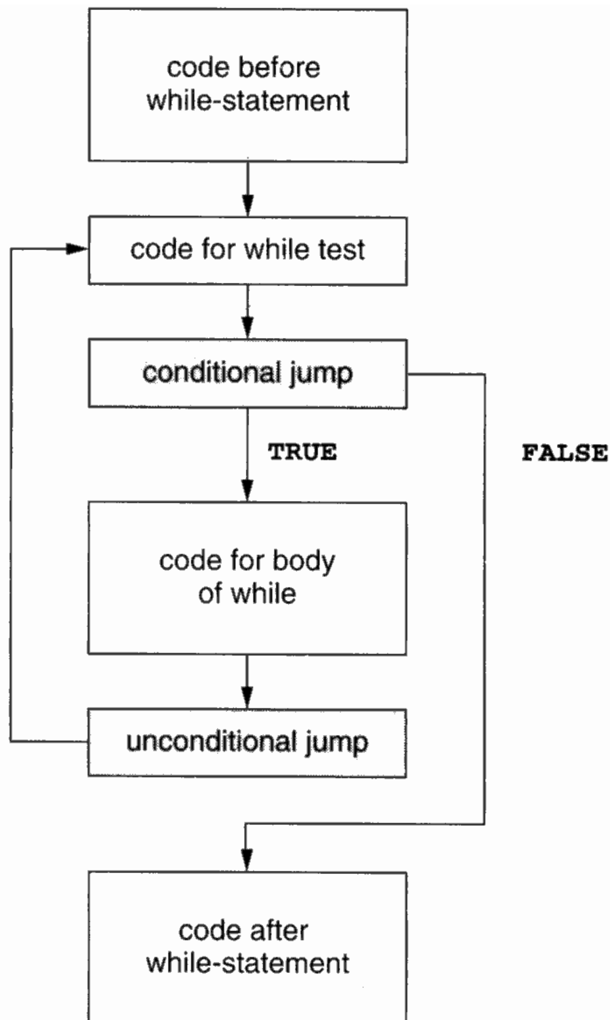
Skisse av TA-kode

```
<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
```

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'

Eksempel i C – siste del beregnes bare dersom første del er sann:

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

a **and** b = **if** a **then** b **else false**

a **or** b = **if** a **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
lod FALSE
lab L2
```

(*a*: hopp fra `fjp L1` til `lab L1`; *b*: hopp fra `ujp L2` til `lab L2`)

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" kunne fjernes (men vi har en :
fjp s2
for selve if-testen. Den måtte med i alle tilfelle)

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**

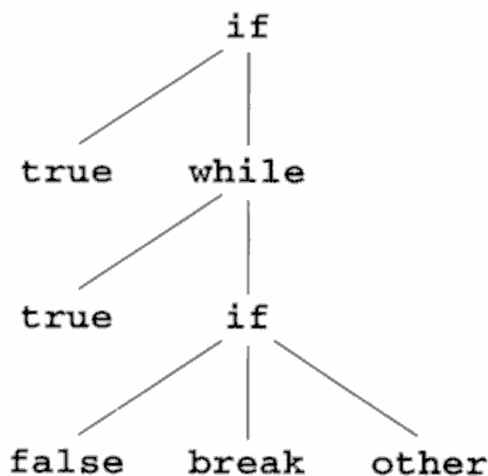
Tre-node:

```
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;
```

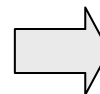
Angir bare false eller true

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

binder 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
```