



Kap. 8 – kodegenerering

INF5110 – 21. april, 2009

Stein Krogdahl,
Ifi UiO

Forelesninger framover:

Torsdag 23. april: 1 time kapittel 8, 1 time Oblig 2

Tirsdag 28. april: Oppgaver kapittel 7, kapittel 8 +



Pensumoversikt - kodegenerering

Helt sikkert pensum:

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 generering for kontroll-setninger og logiske uttrykk

Kanskje også noe fra:

8.5, 8.9 og 8.10

Samt:

- Oppkopierte: Om lur bruk av registre til å holde data som snart skal brukes om igjen (samme som i fjor?)
- Noe om generiske mekanismer i programmeringsspråk
- Litt mer om Javas byte-kode

NB: Detaljert pensumliste kommer!



Hvordan er instruksjonene i en virkelig CPU?

- Ofte: Et antall Registerne (8-128) hvor add, mult, sub, shift ... går mellom disse
- Men på noen maskiner også disse operasjoner 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 (typisk 20 mikro-instr. kan være under utførelse samtidig)
 - "Spekulativ" utføring:
 - Starter på de neste instruksjoner, men må "restarte" dem om inneværende instr. forandrer deres input-data
 - Ved betingede hopp: Utfører 'begge' grener, men gir opp den ene når valget er klart
- Å hente data fra lageret er det som tar tid. Derfor:
 - Flere nivåer med cache-lager, som fylles og utnyttes automatisk.

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

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

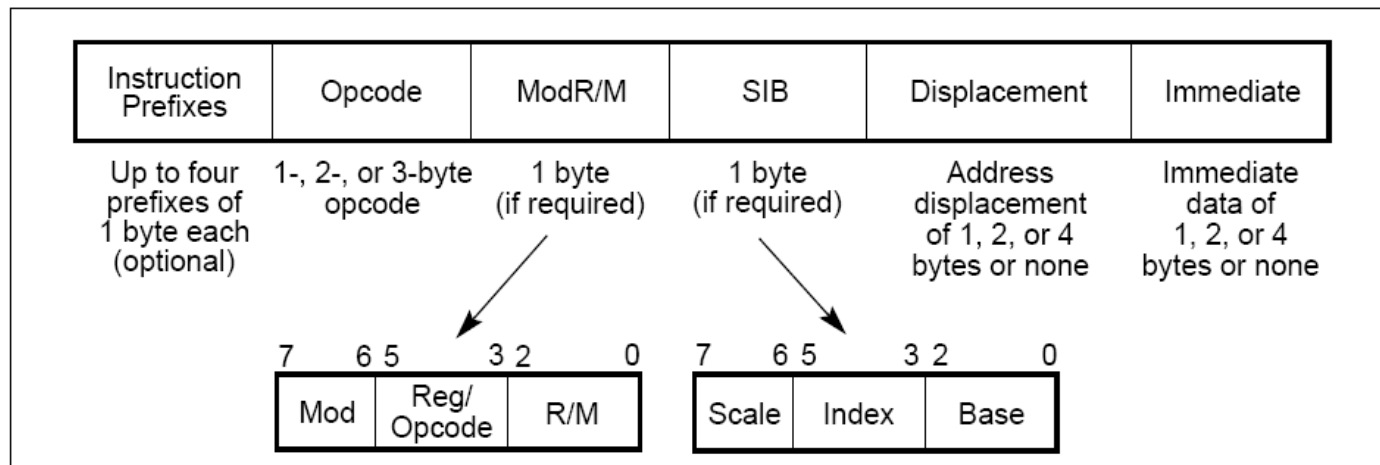


Figure 2-1. IA-32 Instruction Format

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-nivåer – størrelse og hastighet

Lagertype	Størrelse	Rel. hastighet
Register	8 - 128	1 (nå ca. 0.5 - 0.2 ns)
cache L1 data og instr.	12-512 kb	2 - 3
cache L2	0.5-2Mb	10 - 15
cache L3 (Itanium)	noen Mb	?
hoved-memory	2 - 8 Gb	100 - 200
Disk	100-1000 Gb	20 000 000 (ca. 10 ms)

- Utførelsestiden blir vanskelig å forutsi på grunn av cachene
- Det kreves masse logikk for å få dette til å virke (assosiativ aksess)
- Om det er flere "kjerne" på CPUen blir det ekstra ille (lager-koherens)



8.1 Bruk av mellomkode

- Man kan veldig godt generere maskinkode direkte fra syntaks-treet
- Men: Det kan være greit å overføre programmet til en lineær form: "mellom-kode"
 - Grunn: Greit for optimalisering (flytte rundt), greit å legge på fil
- Vi skal se på to former 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, f.eks.:
 - Bevarer vi symboltabellene?
 - Er det operasjoner for array-aksess (eller blir dette løst opp i flere, enklere operasjoner?)

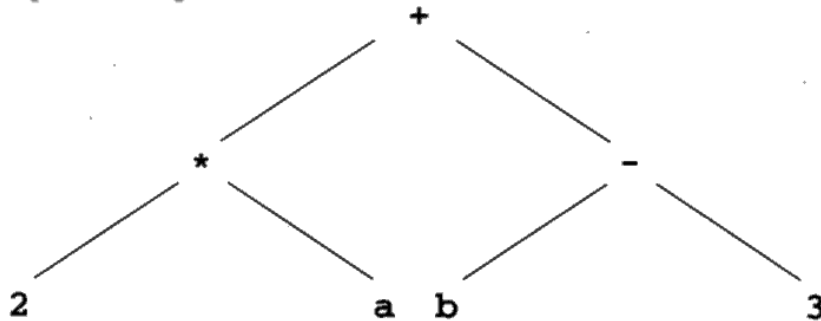


Vi skal se på en del oversettelser:

- 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
 - 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
- Dette kan også sees som eksempel på enkel generering av maskinkode
 - Men da uten å ta hensyn til "registerallokering"

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, \dots er temporære variable

TA grunnform:

$$\mathbf{x = y \ op \ z}$$

op = +, -, *, /, <, >,

and, or

Også:

$$\mathbf{x = op \ y}$$

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

Andre TA-instruksjoner:

$$\mathbf{x = y}$$

if_false x goto L

label L

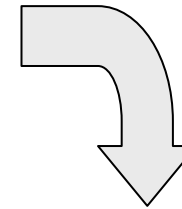
read x

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 "adressen" til instruksjonen er navn på en ny temporær variabel)

En mulig C-struct 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*

op: - opkind (opcode)
addr1: - kind, val/name
addr2: - kind, val/name
addr3: - kind, val/name



P-kode (Pascal-kode – utfører beregning på en stakk)
koden utføres 'normalt' (etter hverandre + jump, **men** beregninger på stakk)

"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")

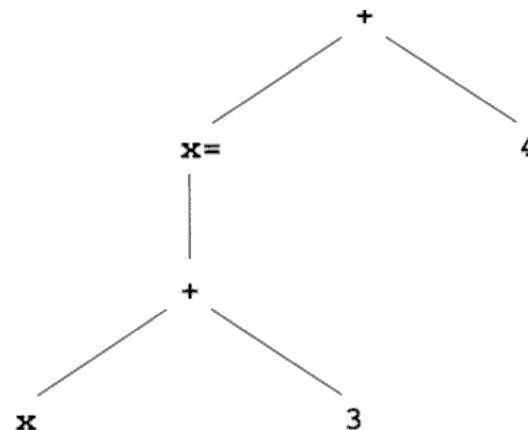
```
1 read x; { input an integer }
2 if 0 < x then { don't compute if x <= 0 }
3   fact := 1;
4   repeat
5     fact := fact * x;
6     x := x - 1
7   until x = 0;
8   write fact { output factorial of x }
9 end
```

```
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
```

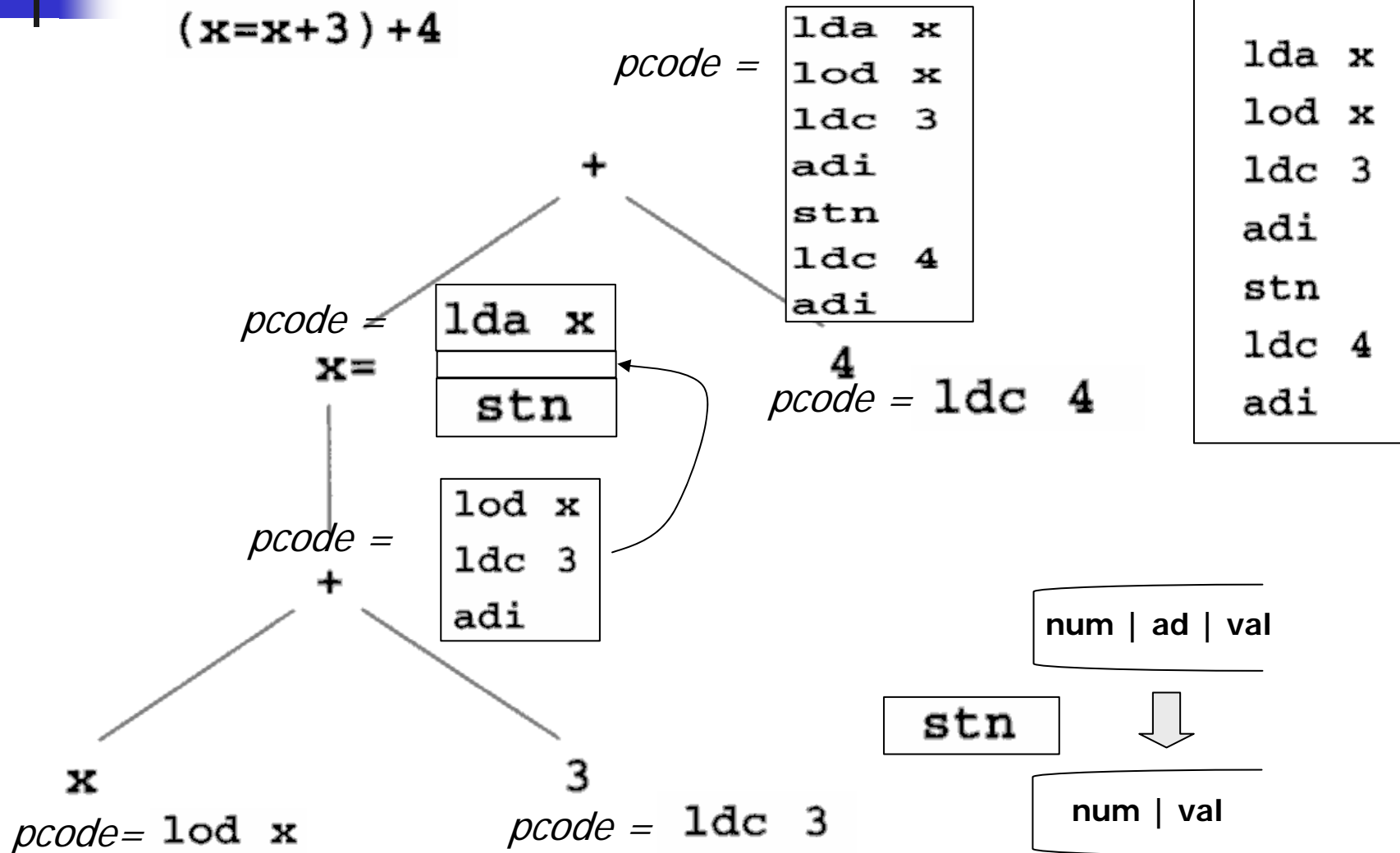
Angivelse av P-kode ved attr.-grammatikk

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

(x=x+3) + 4



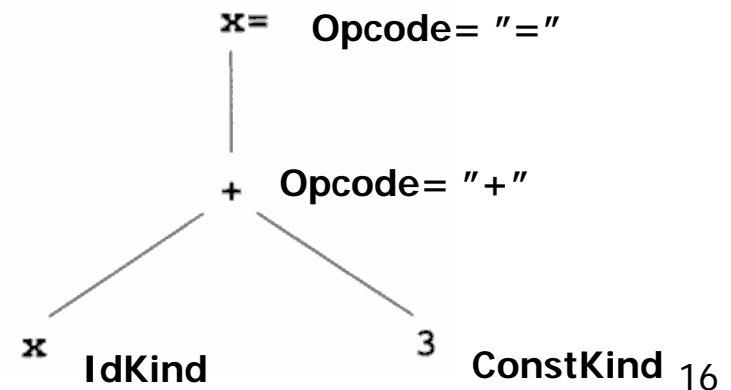
Tenkt generering av P-kode etter attr.-gram. (Vil aldri gjøre det slik i praksis!)



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;
```





Metode-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) ;                                     ← 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;
```

Generert kode (ofte er flere av delene tomme):

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

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 nodeKind {
  case op-node:
    switch opKind {
      case "+": { rek. kall for venstre subtre;
                rek. kall for høyre subtre;
                emit1 ("adi"); }
      case "=": { emit2 ("lda", identifikator);
                 rek. kall for venstre subtre;
                 emit1 ("stn"); }
    }
  case konst-node { emit2 ("ldc", konstant-streng); }
  case ident-node { 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

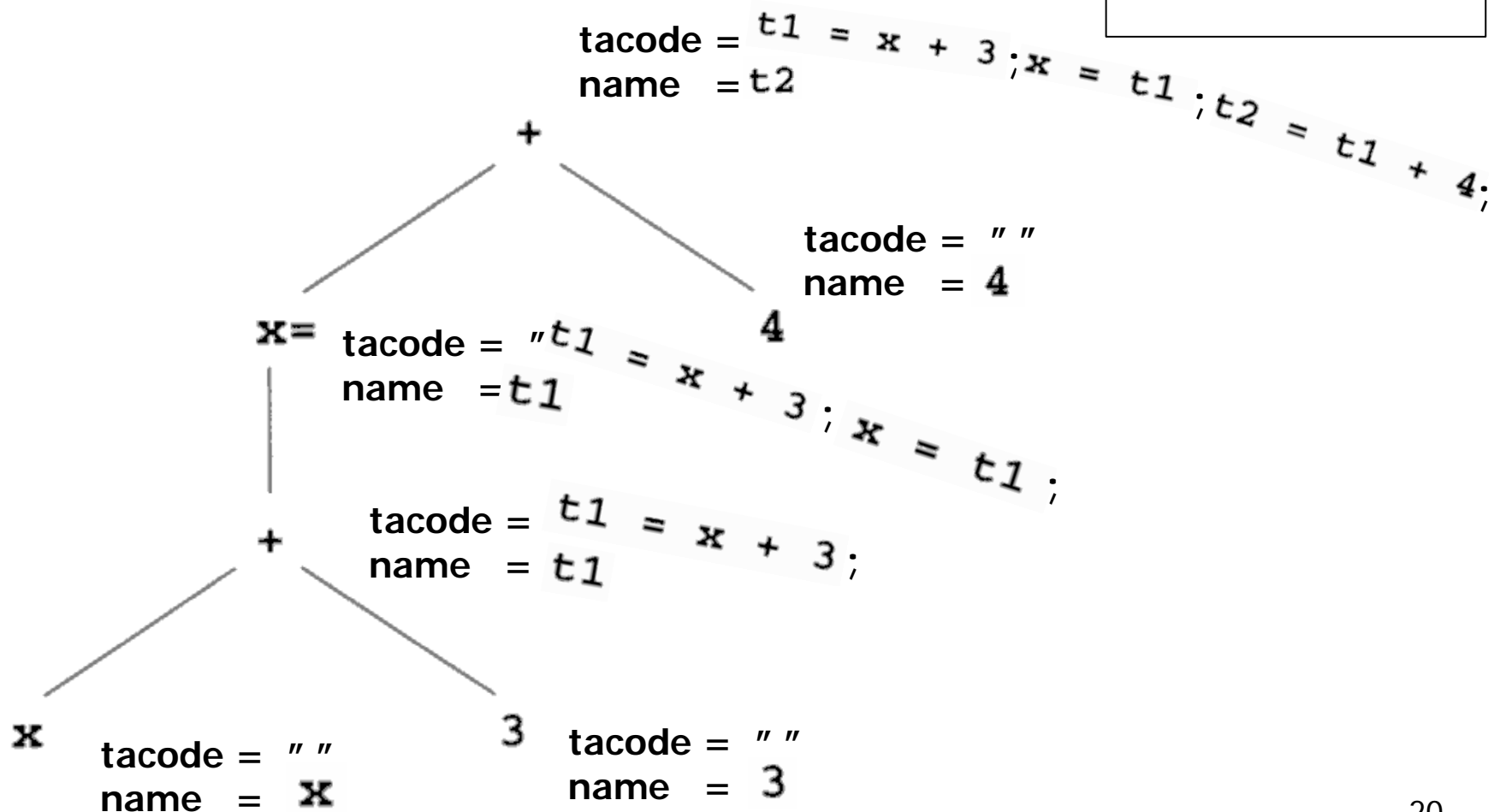
Grammar Rule	Semantic Rules
$exp_1 \rightarrow \mathbf{id} = exp_2$	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode ++$ $\mathbf{id}.strval \parallel "=" \parallel exp_2.name$
$exp \rightarrow aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \rightarrow aexp_2 + factor$	$aexp_1.name = newtemp()$ $aexp_1.tacode =$ $aexp_2.tacode ++ factor.tacode$ $++ aexp_1.name \parallel "=" \parallel aexp_2.name$ $\parallel "+" \parallel factor.name$
$aexp \rightarrow factor$	$aexp.name = factor.name$ $aexp.tacode = factor.tacode$
$factor \rightarrow (exp)$	$factor.name = exp.name$ $factor.tacode = exp.tacode$
$factor \rightarrow \mathbf{num}$	$factor.name = \mathbf{num}.strval$ $factor.tacode = ""$
$factor \rightarrow \mathbf{id}$	$factor.name = \mathbf{id}.strval$ $factor.tacode = ""$

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

(Gjøres ikke slik i praksis)

$(x=x+3) + 4$

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



Generering av rent tekstlig TA-kode fra tre-struktur

Hoved-delen av en rekursive metode. Metoden leverer et navn:

```
switch nodeKind {
  case op-node:
    switch opKind {
      case "+": { tempnavn = nytt temporær-navn;
                opnavn1 = rek kall for venstre subtre;
                opnavn2 = rek kall for høyre subtre;
                emit ("tempnavn = opnavn1 + opnavn2");
                return (tempnavn); }
      case "=": { varnavn = id. for v.s.-variabel (ligger i noden);
                opnavn = rek kall for venstre subtre;
                emit ("varnavn = opnavn");
                return (varnavn); }
    }
  case konst-node { return (konstant-streng); }
  case ident-node { return (identifikator); }
}
```

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

(x=x+3) + 4

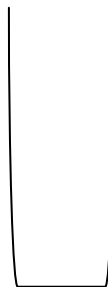
P-kode:

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

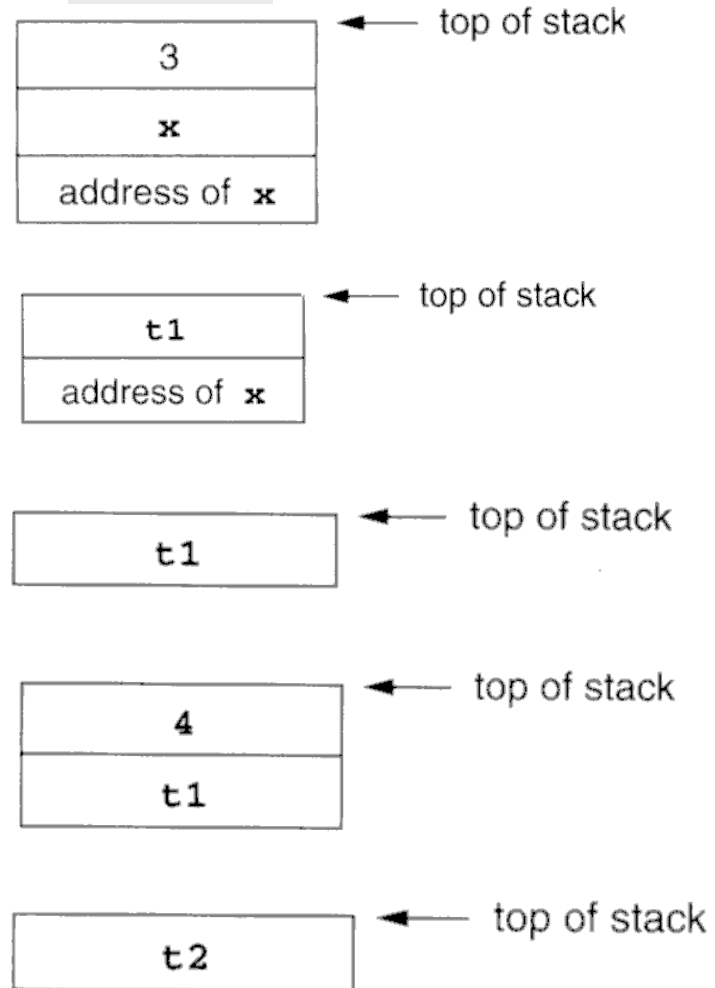
Ønskemål:

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

Stakk:



Stadier:



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

"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
```

Har tidligere sett kortere versjon:

$(x=x+3)+4$

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

$t1 = x + 3$

$x = t1$

$t2 = t1 + 4$

$(x=x+3)+4$

```
lda t1
lod x
ldc 3
adi
sto
lda x
lod t1
sto
lda t2
lod t1
ldc 4
adi
sto
```

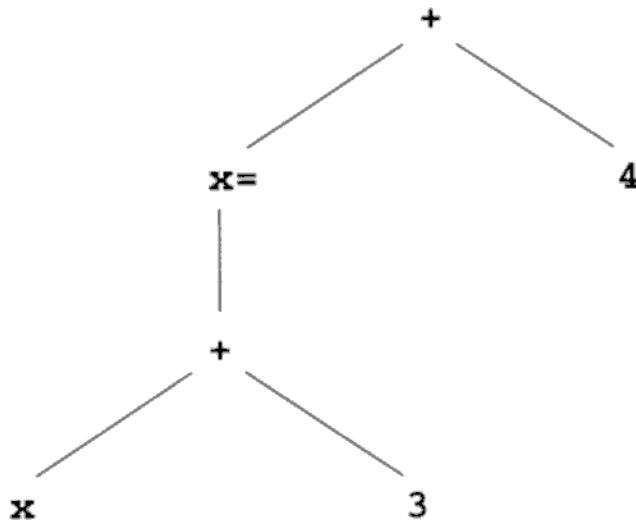
Ny versjon: 13 instr.
Gml. ver. : 7 instr

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

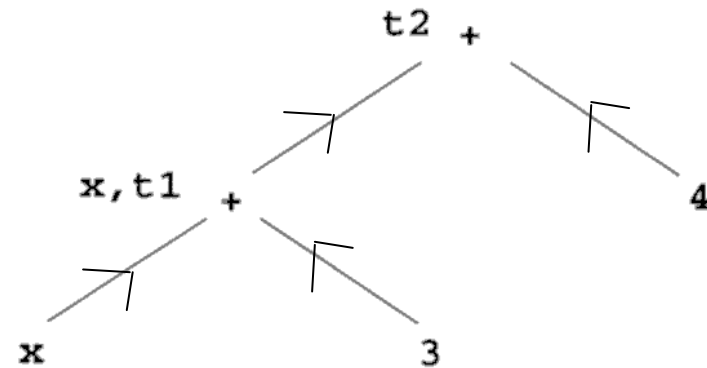
Prøver å lage bedre kode

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

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



Tegner opp "data-flyt" -graf / treet



Generelt: En rettet graf uten løkker (DAG)

Det gjør at det generelt blir vesentlig verre
enn det ser ut her

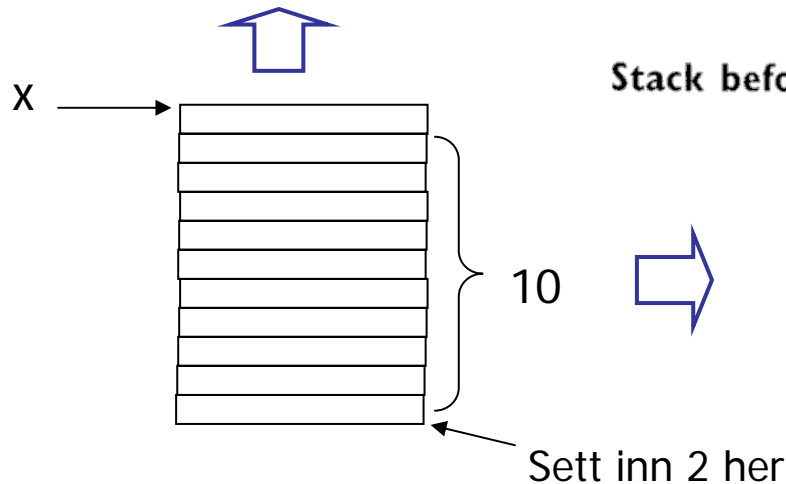
l_{da} x
l_{od} x
l_{dc} 3
a_{di}
s_{tn}
l_{dc} 4
a_{di}

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

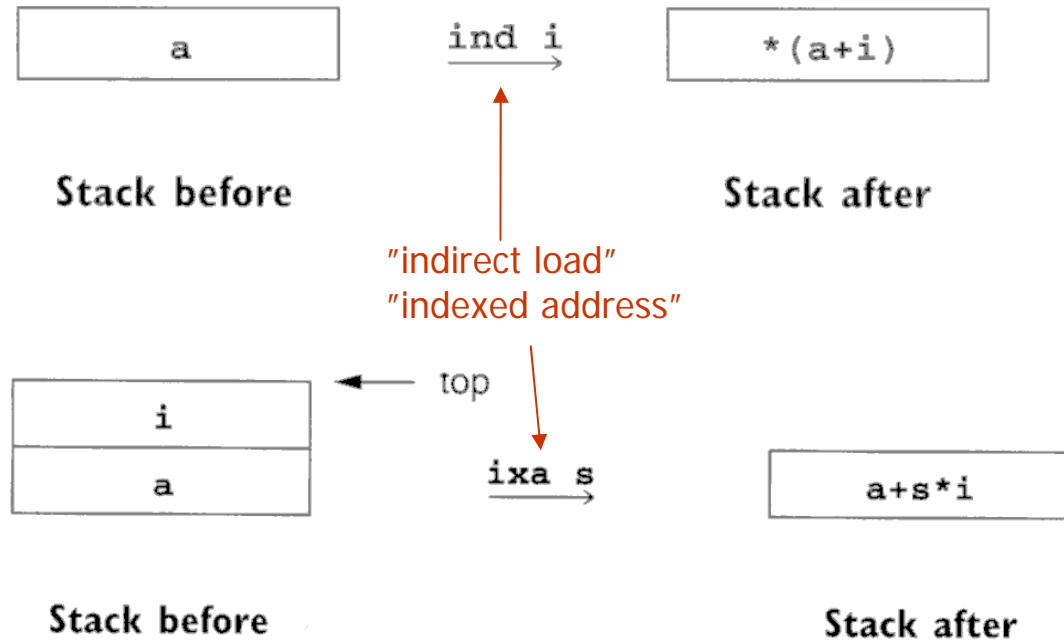
TA-kode:
To nye måter å adressere på

- & X** Adressen til x (ikke for temporære)
- *t** Indirekte gjennom t

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



P-kode: To nye instruksjoner :

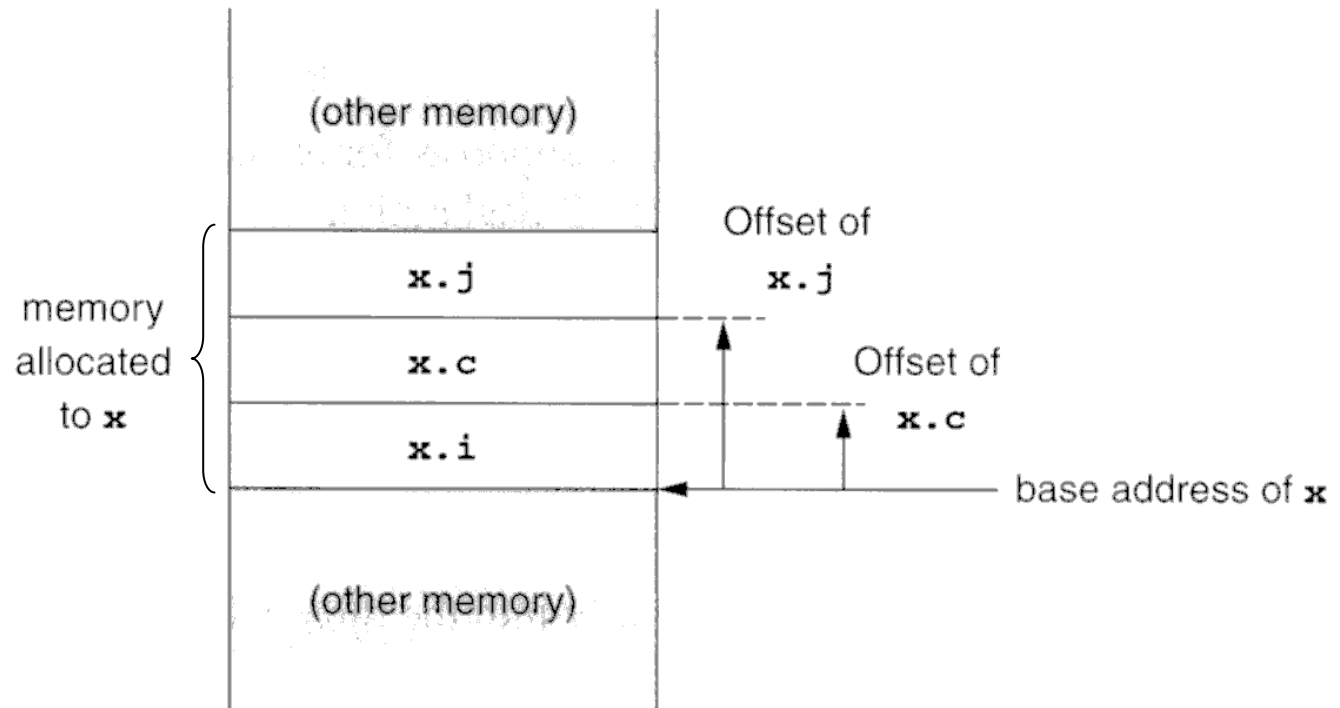


```
lda x
ldc 10
ixa 1
ldc 2
sto
```

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;
```

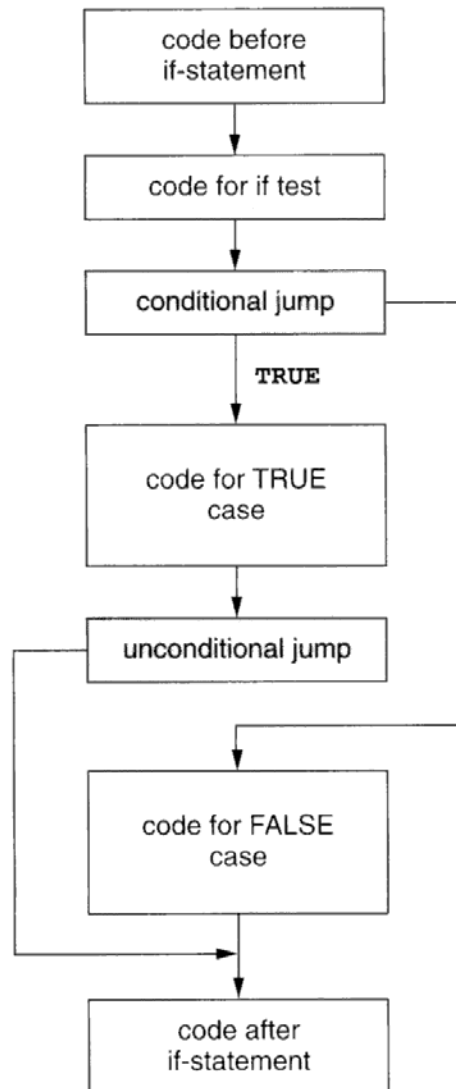




Litt generelt til kap. 8.3

- I boka lages det nokså "lavnivå" TA-kode og P-kode
- Man kan ikke lenger se hva slags språk-konstruksjoner den kommer fra
- Det er ikke opplagt at bokas variant 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 vi får da ikke register-allokerings-problemet

8.4 : If/while – kap.8.3



$if\text{-}stmt \rightarrow \mathbf{if} (exp) stmt \mid \mathbf{if} (exp) stmt \mathbf{else} stmt$
 $while\text{-}stmt \rightarrow \mathbf{while} (exp) stmt$

$\mathbf{if} (E) S1 \mathbf{else} S2$

Skisse av TA-kode

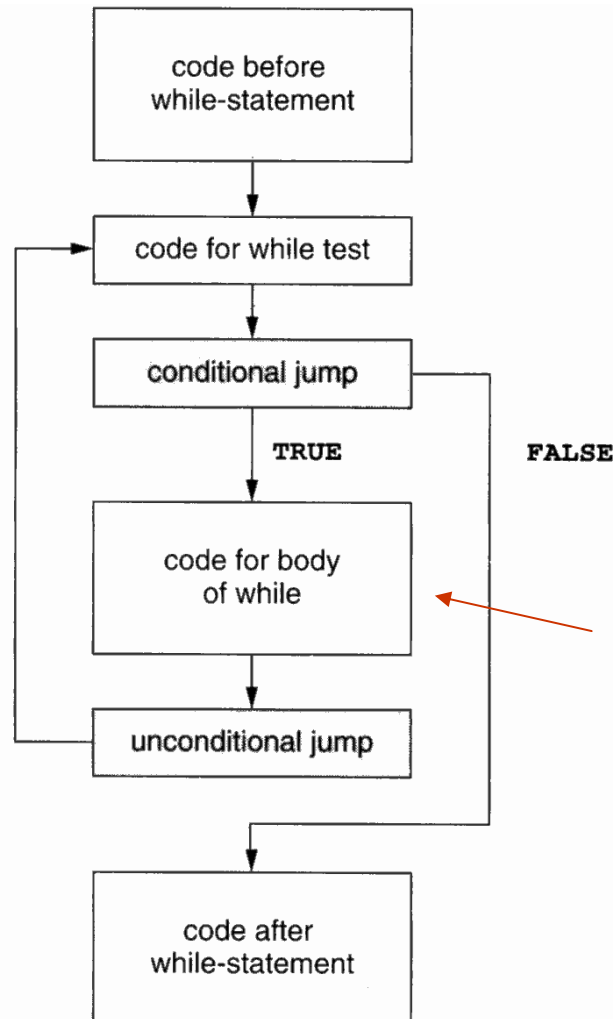
```
<code to evaluate E to t1>  
if_false t1 goto L1  
<code for S1>  
goto L2  
label L1  
<code for S2>  
label L2
```

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
```

Om det skjer et "break" inne i her skal det hoppes ut av while-setningen (til L1)

Behandling av boolske uttrykk

- 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)) ...
```

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

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

`(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 (bracket from `neq` to `lab L1`)

b (bracket from `ujp L2` to `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

Trykkfeil i boka

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
    
```

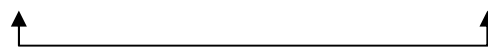
Tre-node:

```

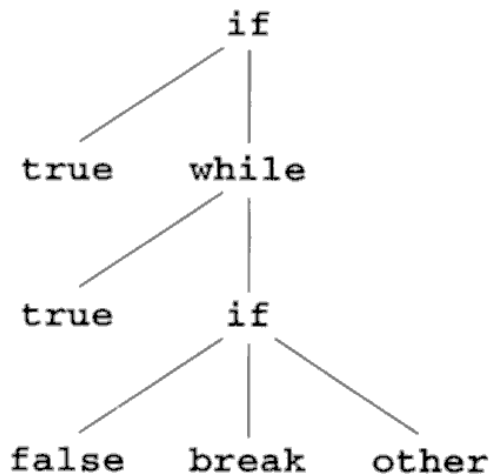
typedef enum {ExpKind,IfKind,
             WhileKind,BreakKind,OtherKind} NodeKind;
typedef struct streenode
{ NodeKind kind;
  struct streenode * child[3];
  int val; /* used with ExpKind */
} STreeNode;
typedef STreeNode *SyntaxTree;
    
```

Angir bare false eller true

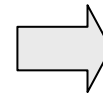
if(true)while(true)if(false)break else other



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
    
```

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

Hit skal en "break" i kildeprogr. gå

```
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);
      Kode for S
      genCode(t->child[1], lab2); 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.



```
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
        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)
      ...
    }
  }
}
```

Hit skal en "break" i kildeprogrammet gå

Om man kommer til "break" skal man gå ut av nærmeste omsluttende while-setning (angitt av label-parameteren)