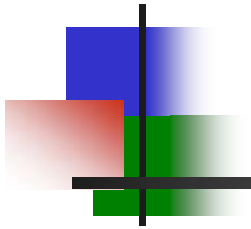


# Kap. 8 del 1 – kodegenerering

## INF5110 – 22. april, 2008

Stein Krogdahl,
Ifi UiO

**Forelesninger framover:**

Torsdag 24 april: *Ikke* forelesning
Tirsdag 29. april: Vanlig forelesning
Torsdag 1. mai: Fridag
Tirsdag 6. mai: Vanlig forlesning

# Pensumoversikt - kodegenerering

**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 generereing for kontroll-setninger og logiske uttrykk**

**( 8.5 Kode-generering for prosedyrer og kall )**

--------- ( Ikke pensum ) ------

8.6 Kode produsert av to kommersielle kompilatorer

8.7 TM: En enkel maskin

8.8 Kode-gen for Tiny på TM

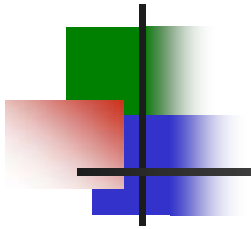
**8.9 og 8.10 Optimalisering (kanskje noe)**

**Også 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 fra Drage-boka blir kopiert opp av kursledelsen og blir delt ut (eller kan hentes).



# 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.

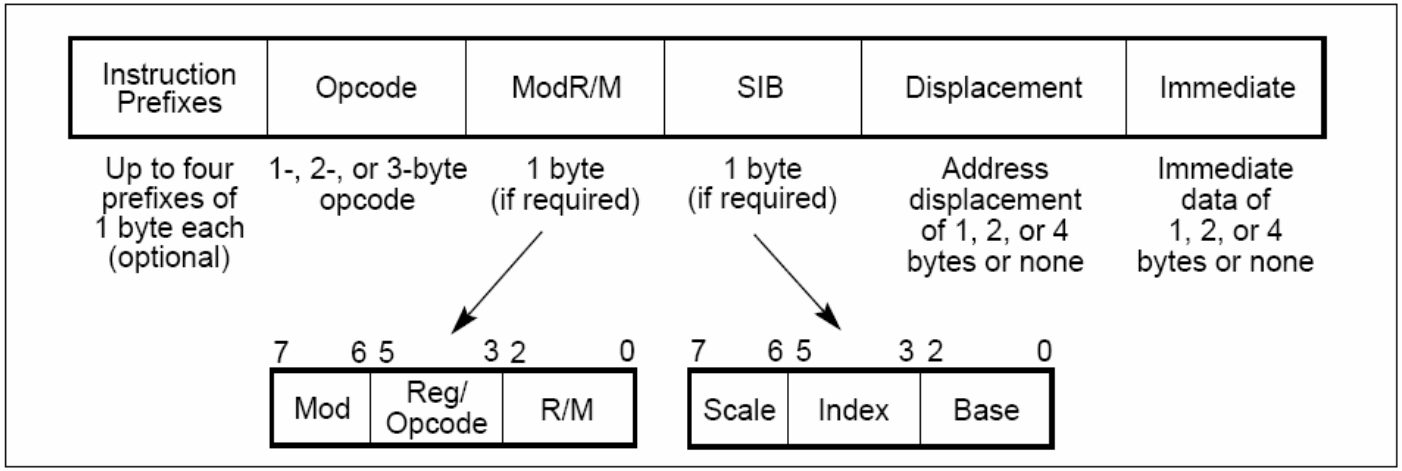



**Figure 2-1. IA-32 Instruction Format**

### 5.1.2 Binary Arithmetic Instructions

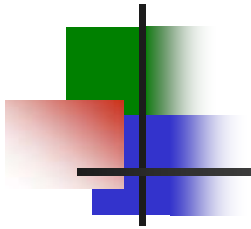
The binary arithmetic instructions perform basic binary integer computations on byte, word, and doubleword integers located in memory and/or the general purpose registers.

| | |
|---|---|
| ADD | Integer add |
| ADC | Add with carry |
| SUB | Subtract |

# Hukommelses-nivåene – størrelse og hastighet

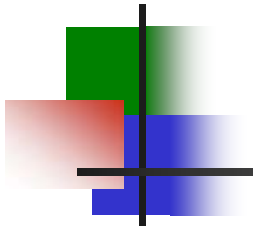
| **Lagertype** | **Størrelse** | **Rel. hastighet** |
|---|---|---|
| Register | 8 - 128 | 1 (nå ca. 0.7-0.3 ns) |
| cache L1<br>data og instr. | 12-512 kb | 2-3 |
| cache L2 | 0.5-2Mb | 10-15 |
| cache L3<br>(Itanium) | noen Mb | ? |
| hoved-memory | 2 - 8 Gb | 100 |
| Disk | 80-800 Gb | 20 000 000 (ca. 10 ms) |

NB: Om det er flere kjerner på CPUen kan de gå i beina på hverandre



# 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 de løst opp i flere, enklere operasjoner?)



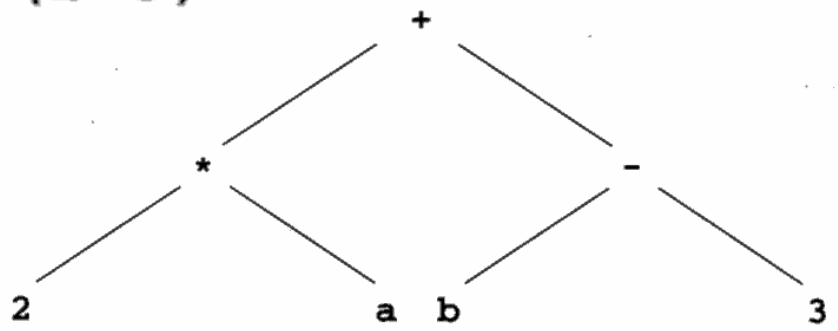
# Vi skal se på en del oversettelser:

- 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
    - 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

- Pensum fra annen bok (Ahu, Sethi og Ullmann):
  - Dette deles ut ferdig kopiert opp
  - Har definisjon av en litt forenklet register-maskin
  - Oversettelse fra TA-kode til denne maskinkoden, med noen typiske problemstillinger.

# 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**

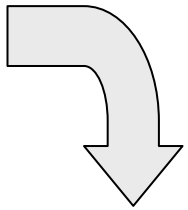
op = not, -, float-to-int. ...

Andre TA-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?

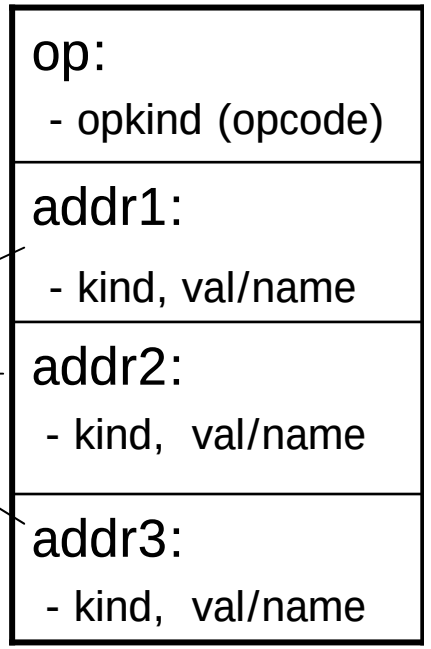
- -kvadrupler
- -tripler der også ”adressen” er navn på en temporær

# 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*

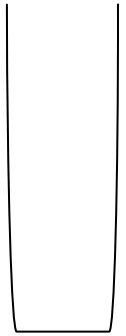


# P-kode (Pascal-kode – utfører beregning på en stakk) - del I

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

**"push" koder (= load på stakken):**

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



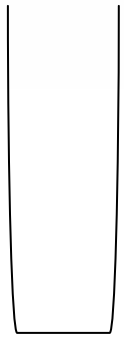
`2*a+(b-3)`

```
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”)

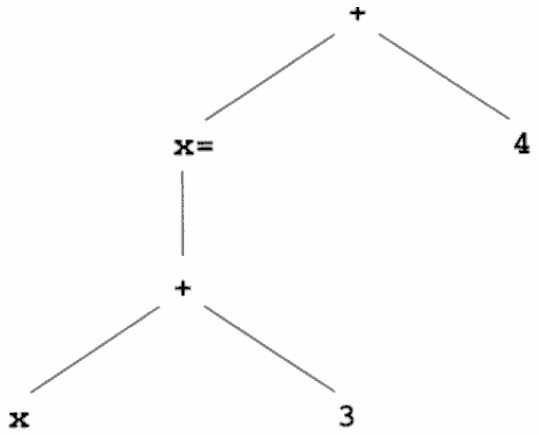
```
1  read x; { input an integer }
2  if 0 < x then { don't compute if x <= 0 }
3    fact := 1;
4    repeat
5      fact := fact * x;
6      x := x - 1
7    until x = 0;
8    write fact { output factorial of x }
9  end
```

```
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
```

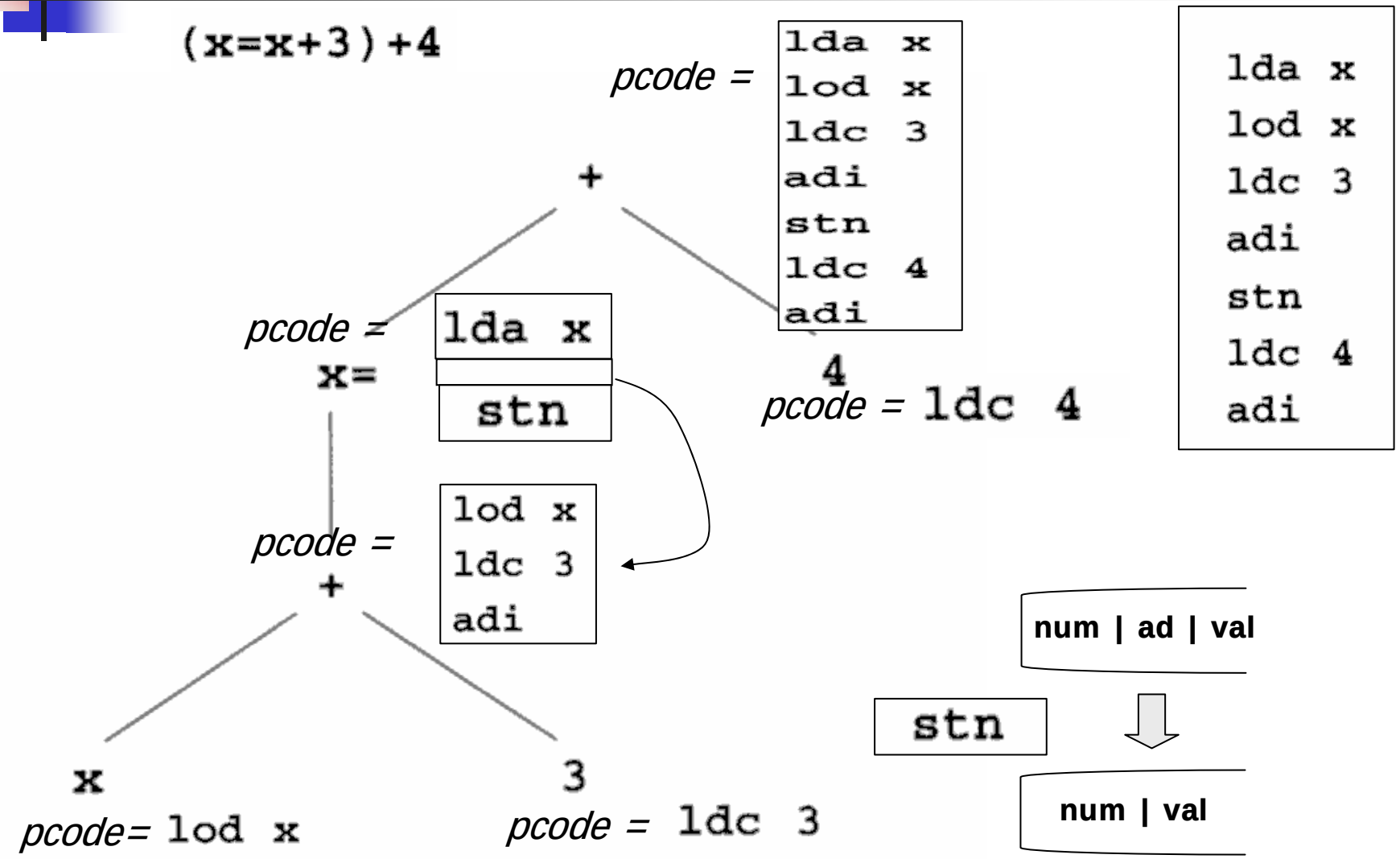
# Angivelse av P-kode ved attr.-grammatikk

| Grammar Rule | Semantic Rules |
|---|---|
| $exp_1 \rightarrow$ **id** = $exp_2$ | $exp_1$ .pcode = "**lda**" \|\| **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 attr.-gram.



# Angivelse av TA-kode ved attr.-grammatikk

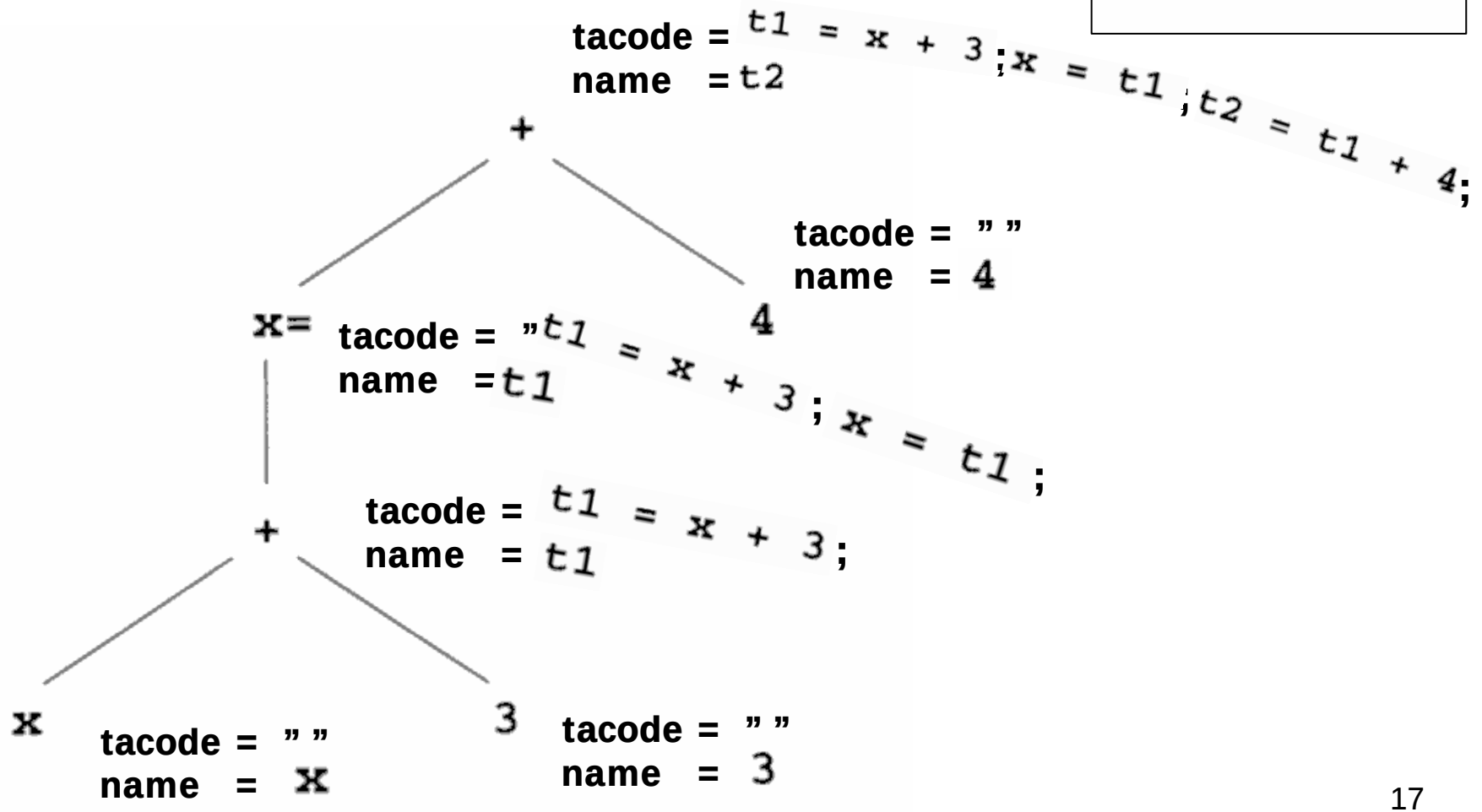
`(x=x+3)+4`

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

# Generering av TA-kode etter attr.-gram.

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

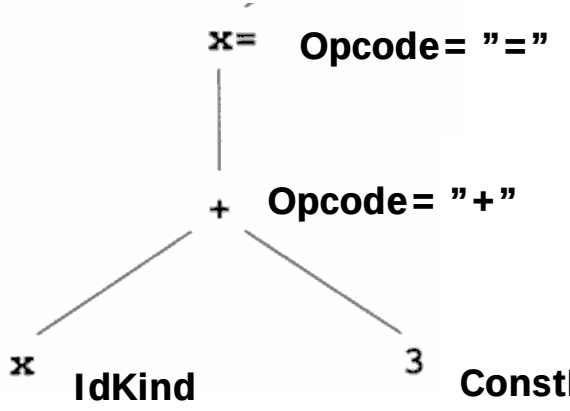
(x=x+3)+4



# Kodegenerering kan gjøres ved rekursiv gjennomgang av syntakstreet. 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;
```



# 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 ;
      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**

← **rek kall**

← **Infiks - operasjoner**

← **rek kall**

← **Postfiks - operasjoner**

# 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;
  }
}
```

# Generering av rent tekstlig TA-kode fra tre-struktur

## Rekursiv metode, 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); }
}
```

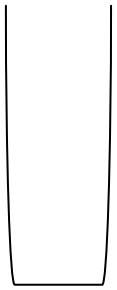
# Fra P-kode til TA-kode ("Statisk simulering")

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

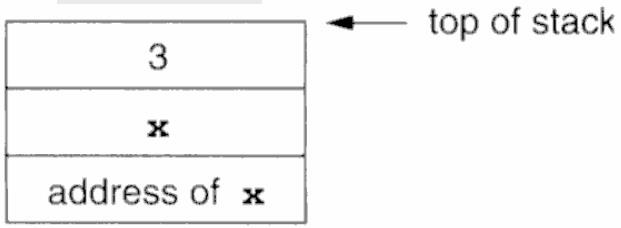
P-kode:

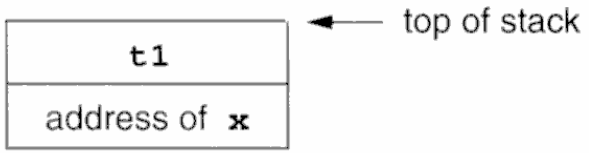
```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

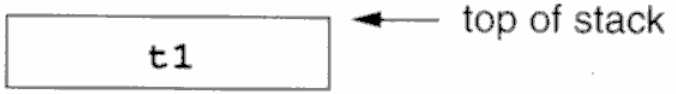
Stakk:

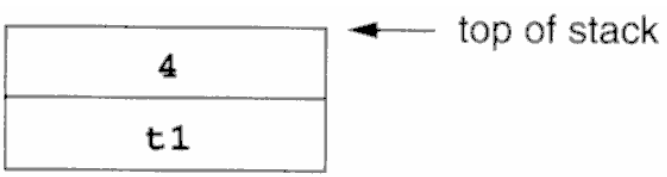


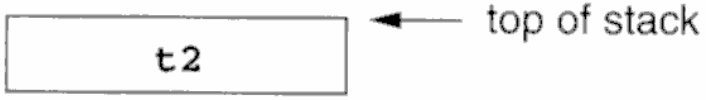
Stadier:




Ønskemål:

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

# 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
```

```
(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
```

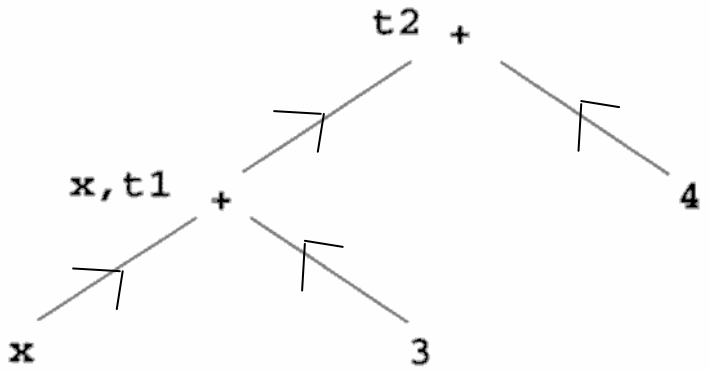
Ny versjon: 13 instr.
Gml. ver. : 7 instr

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

Prøver å lage bedre kode

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

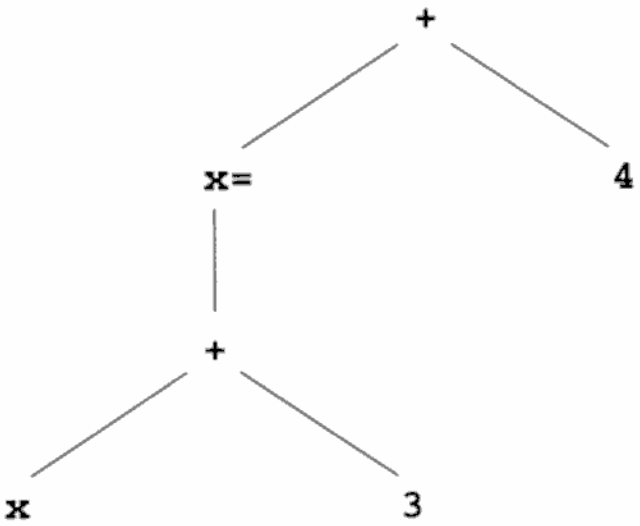
Tegner opp ”data-flyt” –grafen / treet

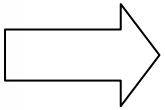



Generelt: En rettet graf uten løkker (DAG)

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

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



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

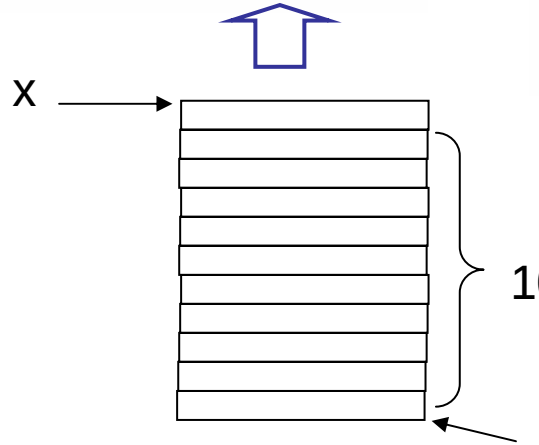
# Kap. 8.3 – Aksess av datastruktur, trenger mer fleksibel adresse-beregning

TA-kode:
To nye måter å adressere på

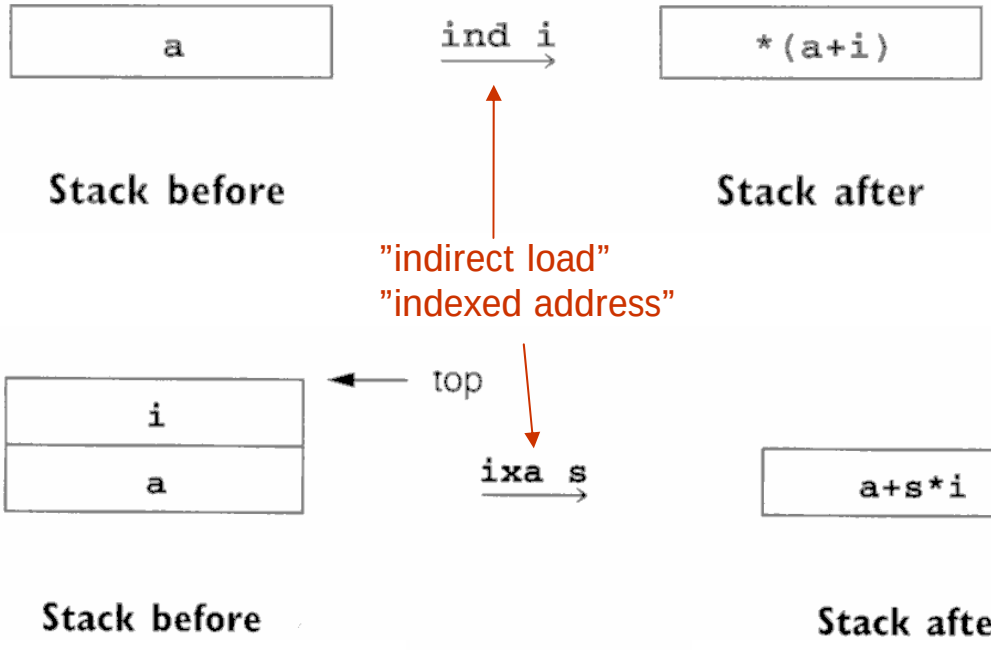
**& X** Adressen til x
(ikke for temporære)

***t** Indirekte gjennom t

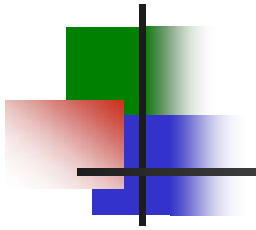
```
t1 = &x + 10
*t1 = 2
```




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 vi får da ikke register-allokerings-problemet