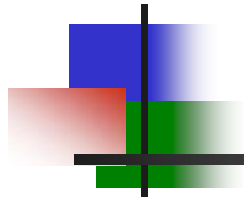


INF 5110, 1. februar 2012

Stein Krogdahl



---

## Dagens Tema: Grammatikker

### Kap. 3 i K. C. Louden

Min Foil-stil: Ofte mer tekst enn man helt kan få med seg på forelesningen, for at de skal være gode til repetisjon

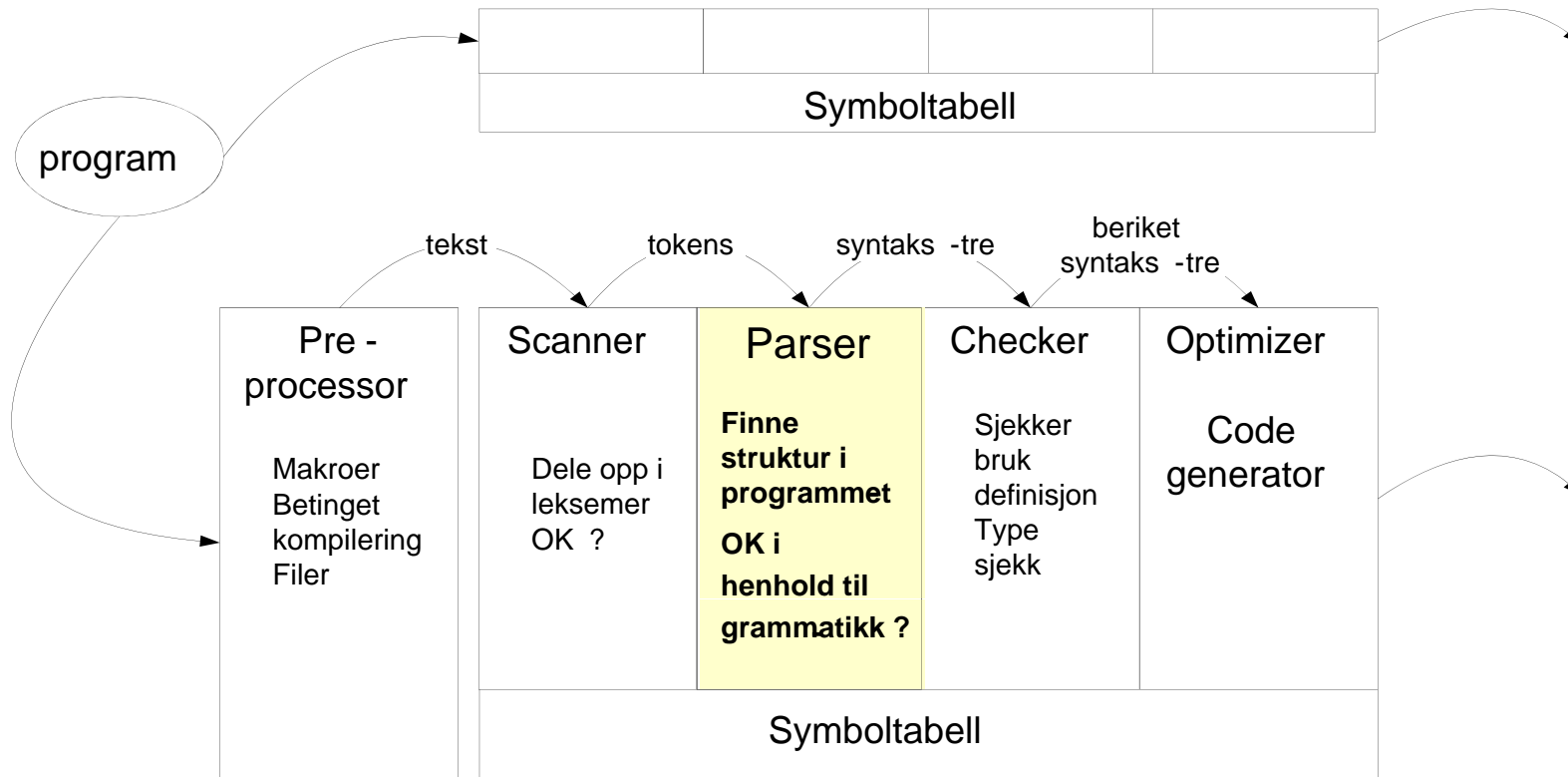


# Stein Krogdahl:

---

- Utdannet:
  - Ved UiO, Cand.Real. 1973
  - I "Databehandling", som da lå under Matematisk Institutt
- Har vært ved:
  - Universitetet i Tromsø (1973 – 78)
  - Norsk Regnesentral (1978 – 82)
    - Laget Simula-kompilator med Birger Møller-Pedersen
  - Universitetet i Oslo, Ifi (1982 – nå)
- Har drevet med
  - Kombinatoriske algoritmer (underviser noe av INF 4130):
    - Finn beste utplukk som tilfredsstillir bestemte krav o.l.
  - Formell verifikasjon av programmer:
    - Helst resonnementer som ikke-eksperter kan følge
  - Programmeringsspråk: Design og implementasjon
  - Nå: Mest SWAT-prosjektet (om modulariserings-mekanismer i programmeringsspråk)

# Hvor er vi nå - kap. 3, 4 og 5:



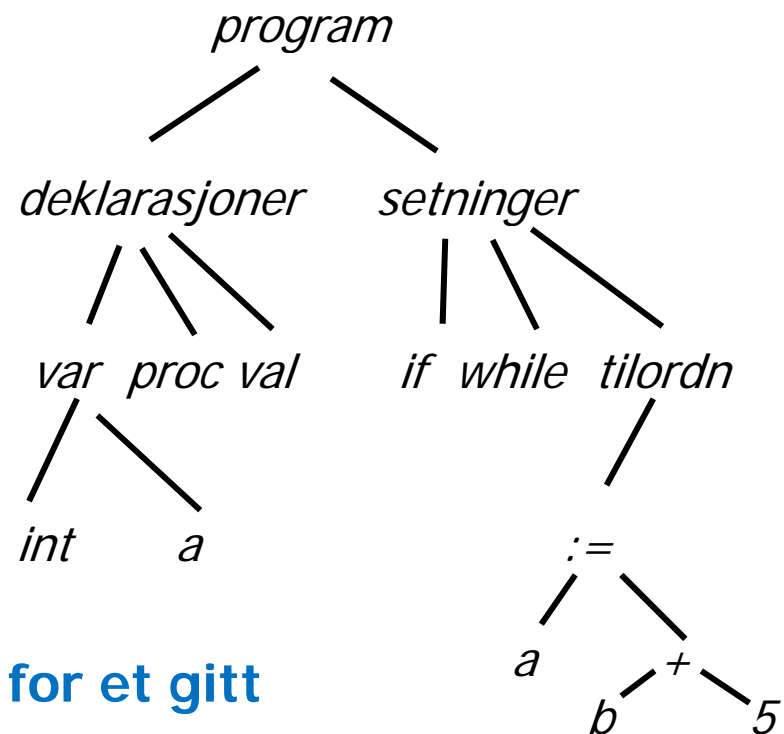
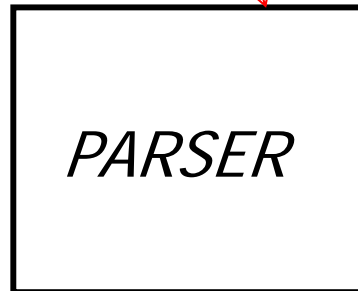
Lex  
Flex  
Grammatikker.  
"Top-down"- og  
"bottom-up"-parsering.  
**Verktøy:** Antlr, Yacc,  
Bison, CUP, m.fl.

Attributtgrammatikker  
+  
div. mer eller mindre  
systematiske metoder

# Forenklet skisse av hva en parser gjør

Sekvens av  
Token  
(leksemer) fra  
scanner

*Sjekker også at token-  
sekvensen utgjør et  
syntaktisk riktig program.  
Om feil: Skriv **forståelig**  
feilmelding!*



**Syntaks-tre for et gitt  
program:**

"Abstrakt" eller "konkret"?

Dette treet er typisk **abstrakt**.



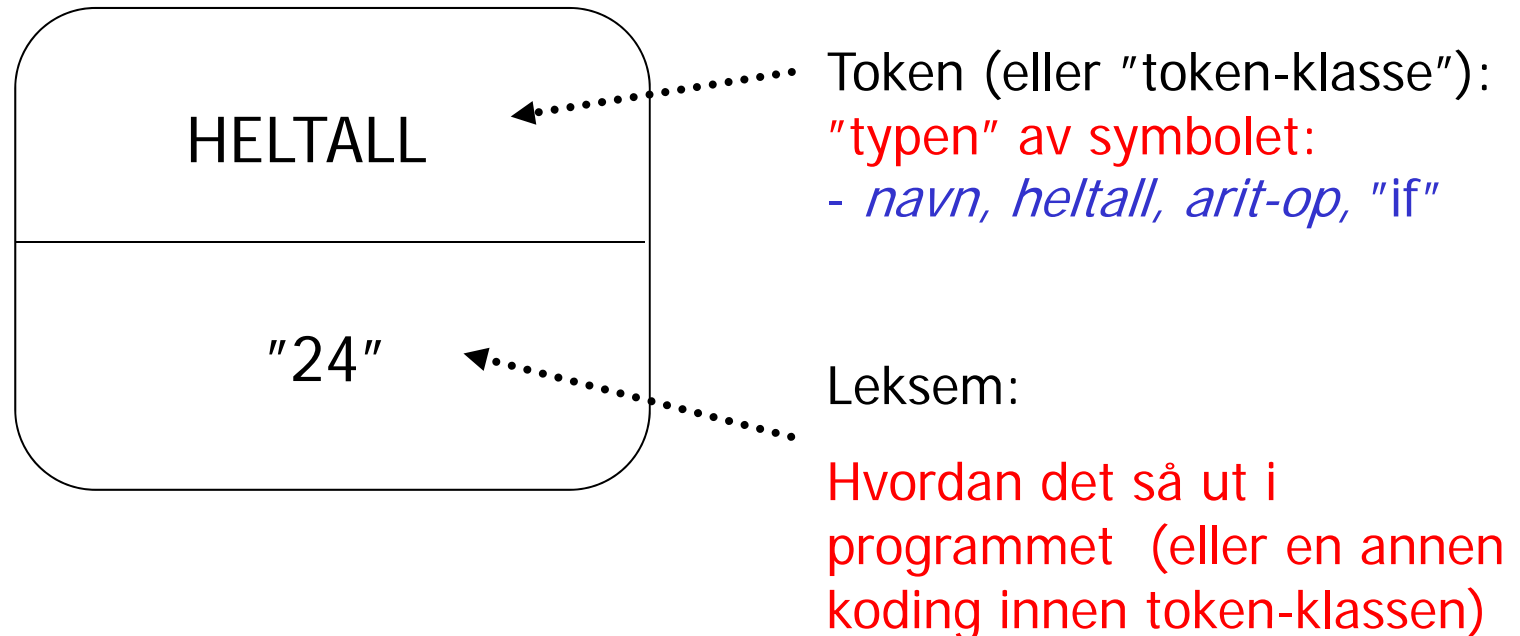
## Oversikt – kap 3 (grunnleggende om grammatikker)

---

- Kontekstfrie grammatikker og BNF-notasjon
- Hvordan definerer en grammatikk et språk?
- Parserings-trær og abstrakte syntaks-trær
- Entydige/flertydige grammatikker
- Utvidet notasjon: EBNF og syntaksdiagrammer
- Eksempel
  - **Tiny**
- **Antakeligvis også i år:** Før vi startet på kapittel 4 tok vi også mer generelt stoff om grammatikker. Dette er i boka er vevd innimellom i kap. 4 og 5.

# Hva får vi fra scanneren

Vi får en sekvens av slike (der kommentarer, linjeskifte etc. er vekk):



- Ofte kalles også det *hele* for et "token",
- I forbindelse med parsering kalles det også et "terminalsymbol"
- Oppdelingen i token-klasser er ikke alltid opplagt? (+ og \* sammen?)



## Om BNF-grammatikker:

---

- En grammatikk bruker et antall **symboler**, for å beskrive de setninger (programmer) som er med i et "språk"
  - **Terminal-symboler** – De vi får fra scanneren, og oftest tenker vi da mest på deres token-type: *navn, heltall, "if", ...*
  - **Ikketerminal-symboler** - WHILE-SETNING, TILORDNING, KLASSEDEKL, UTTRYKK, ...
    - Startsymbolet : Lovlige setninger er avledet fra dette.
  - **Meta-symboler** - Hjelpesymboler/tegn vi bruker for å sette opp reglene.
- **Merk:** En grammatikk egner seg best til å *lage* (avlede) riktige setninger ut fra startsymbolet
  - Parserings-problemet er det **omvendte**:
    - Gitt en setning. Kan denne avledes i grammatikken, og hvordan?
- En grammatikk spesifiserer et språk via *regler* for lovlige sammensetninger av terminal og ikke terminalsymboler
  - Reglene kalles også *produksjoner*

# Kontekstfrie grammatikker: BNF-notasjon med variasjoner

- BNF = Backus (Fortran) – Naur (Algol) – Form

- Bokas vanlige notasjon:

$exp \rightarrow exp \ op \ exp \mid (exp) \mid \mathbf{number}$

$op \rightarrow + \mid - \mid *$

- Metasymboler: " $\rightarrow$ " (leses: "Kan ha formene") , "|" (leses: "eller")
- Ikke-terminaler:  $exp, op$
- Terminaler :  $\mathbf{number, (, ), *, +, -}$
- Startsymbol:  $exp$

- En tradisjonell måte (Algol 60 rapporten):

$\langle exp \rangle ::= \langle exp \rangle \langle op \rangle \langle exp \rangle \mid ( \langle exp \rangle ) \mid \mathbf{NUMBER}$

$\langle op \rangle ::= + \mid - \mid *$

*Metasymbol*

*Terminal-symbol*

- Utvidet BNF (EBNF): Merk at man må være nøye med parenteser!

$exp \rightarrow exp ( "+" \mid "-" \mid "*" ) exp \mid ( " (" exp )" " ) \mid \mathbf{number}$





## Flere måter å skrive den samme grammatikken

- Følgende regnes av boka som den mest basale formen av BNF:

$$exp \rightarrow exp \ op \ exp$$
$$exp \rightarrow ( \ exp \ )$$
$$exp \rightarrow \mathbf{number}$$
$$op \rightarrow +$$
$$op \rightarrow -$$
$$op \rightarrow *$$

*6 regler eller  
"produksjoner"*

- Kortest mulig (men vi sier gjerne at det fremdeles er 6 regler)

$$E \rightarrow E \ O \ E \mid ( \ E \ ) \mid \mathbf{n}$$
$$O \rightarrow + \mid - \mid *$$

# Avledning (venstreavledning) av: (number - number) \* number

$exp \rightarrow exp\ op\ exp$   
 $exp \rightarrow ( exp )$   
 $exp \rightarrow \mathbf{number}$   
 $op \rightarrow +$   
 $op \rightarrow -$   
 $op \rightarrow *$

## Mellomformer (setningsformer)

## Regel (produksjon) brukt

- |     |   |                                     |
|-----|---|-------------------------------------|
| (1) | $exp \Rightarrow exp\ op\ exp$                                      | $[exp \rightarrow exp\ op\ exp]$    |
| (2) | $\Rightarrow (exp)\ op\ exp$  | $[exp \rightarrow ( exp )]$         |
| (3) | $\Rightarrow (exp\ op\ exp)\ op\ exp$                               | $[exp \rightarrow exp\ op\ exp]$    |
| (4) | $\Rightarrow (\mathbf{number}\ op\ exp)\ op\ exp$                   | $[exp \rightarrow \mathbf{number}]$ |
| (5) | $\Rightarrow (\mathbf{number} - exp)\ op\ exp$                      | $[op \rightarrow -]$                |
| (6) | $\Rightarrow (\mathbf{number} - \mathbf{number})\ op\ exp$          | $[exp \rightarrow \mathbf{number}]$ |
| (7) | $\Rightarrow (\mathbf{number} - \mathbf{number}) * exp$             | $[op \rightarrow *]$                |
| (8) | $\Rightarrow (\mathbf{number} - \mathbf{number}) * \mathbf{number}$ | $[exp \rightarrow \mathbf{number}]$ |

*Vestreavledning = gjør hele tiden videre avledning fra ikke-terminalen lengst til venstre. Ferdig når det bare er terminal-symboler*

*Språket til G:*

$$L(G) = \{ s \mid exp \Rightarrow^* s \}$$

↑  
grammatikk

↑  
streng med bare terminal-symboler

# Avledning (høyreavledning)

av: (number - number) \* number

$exp \rightarrow exp\ op\ exp$   
 $exp \rightarrow ( exp )$   
 $exp \rightarrow \mathbf{number}$   
 $op \rightarrow +$   
 $op \rightarrow -$   
 $op \rightarrow *$

Startsymbol

Produksjon brukt

(1) $exp \Rightarrow exp\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
(2) $\Rightarrow exp\ op\ \mathbf{number}$	$[exp \rightarrow \mathbf{number}]$
(3) $\Rightarrow exp\ * \mathbf{number}$	$[op \rightarrow * ]$
(4) $\Rightarrow ( exp ) * \mathbf{number}$	$[exp \rightarrow ( exp )]$
(5) $\Rightarrow ( exp\ op\ exp ) * \mathbf{number}$	$[exp \rightarrow exp\ op\ exp]$
(6) $\Rightarrow (exp\ op\ \mathbf{number}) * \mathbf{number}$	$[exp \rightarrow \mathbf{number}]$
(7) $\Rightarrow (exp - \mathbf{number}) * \mathbf{number}$	$[op \rightarrow - ]$
(8) $\Rightarrow (\mathbf{number} - \mathbf{number}) * \mathbf{number}$	$[exp \rightarrow \mathbf{number}]$

*Høyreavledning = bruk alltid ikketerminalen lengst til høyre  
Ferdig når det bare er terminalsymboler*

*Det finnes også masse andre måter å avlede samme setning fra grammatikken*



# Opplagte krav til en fornuftig grammatikk

---

Alle terminaler og ikke-terminaler:

- Må kunne inngå i en streng avledet fra startsymbolet

Alle ikketerminaler

- Må kunne avledes videre til noe som bare inneholder terminal-symboler

■ Eks:

$A \rightarrow B x$

$B \rightarrow A y$

$C \rightarrow z$

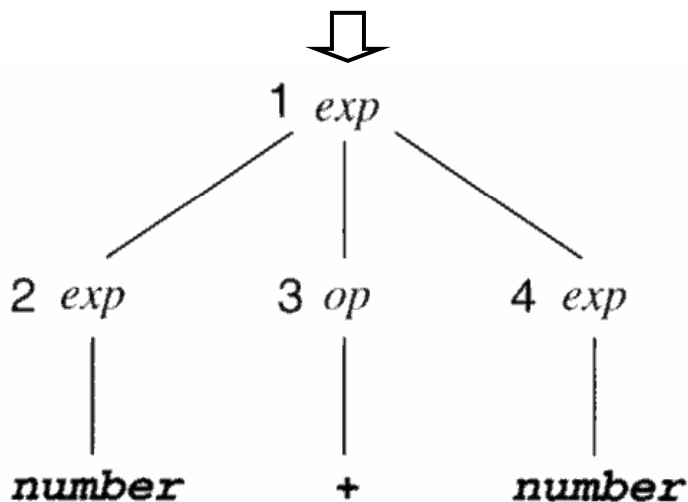
- C eller z kan ikke inngå i noen streng avledet fra A
- Kan aldri avlede noe fra A som bare har terminalsymboler

Altså en *håpløs* grammatikk

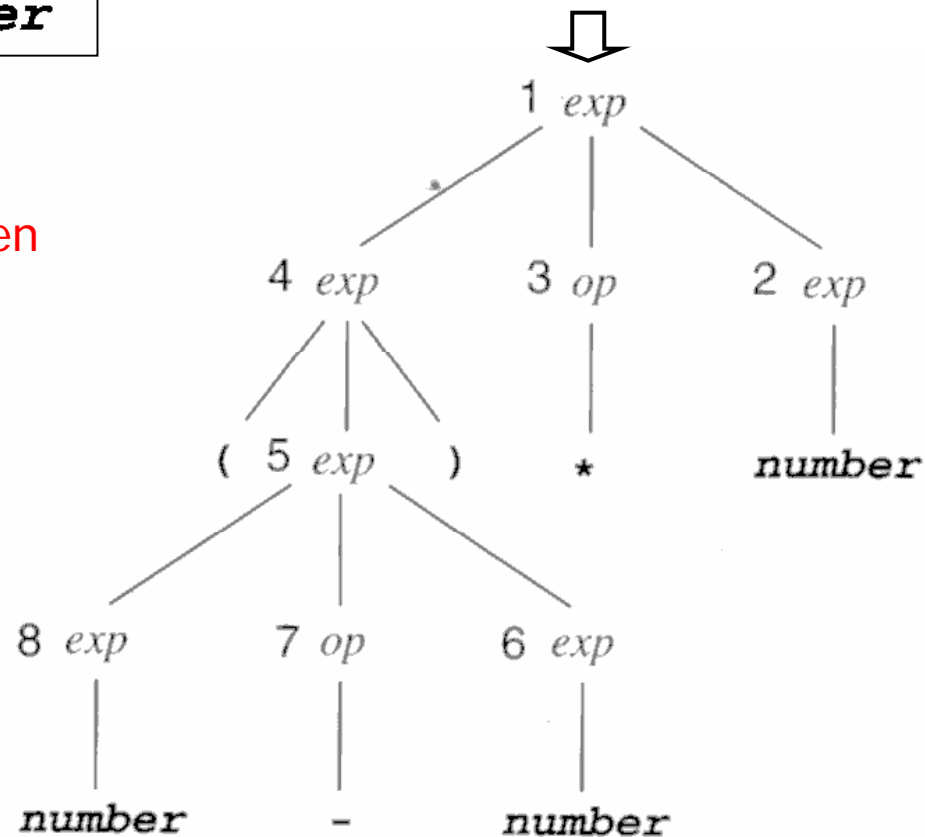
# Parserings-tre (ofte kalt: konkret syntaks-tre)

- (1)  $exp \Rightarrow exp \text{ op } exp$
- (2)  $\Rightarrow \mathbf{number} \text{ op } exp$
- (3)  $\Rightarrow \mathbf{number} + exp$
- (4)  $\Rightarrow \mathbf{number} + \mathbf{number}$

- Exp: **num + num**
- Viktig: En representasjon som er *uavhengig* av avlednings-rekkefølgen
- Tallene angir venstre-avledning

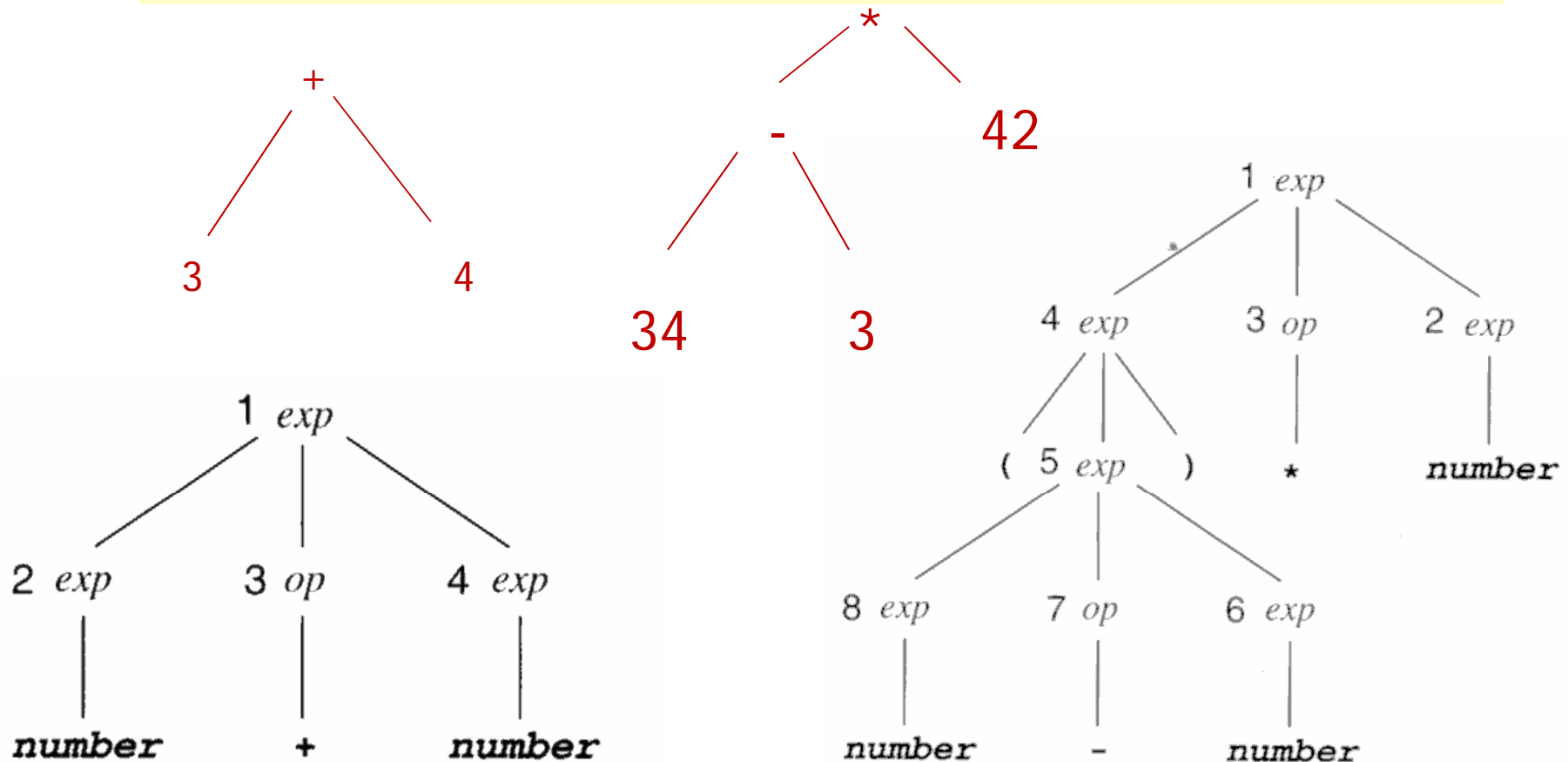


- Exp: **(num - num) \* num**
- Tallene angir høyre-avledning
- Ser vi bort fra tallene, gir altså alle avlednings-rekkefølger det samme treet:



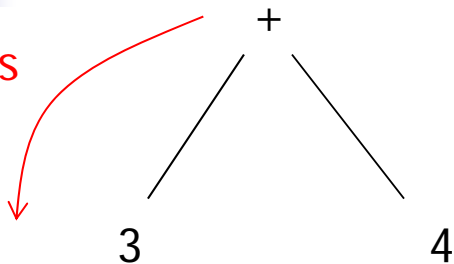
# Abstrakt syntakstre (AST)

Vi tar bort de "unødvendige" nodene i treet, de som stammet fra "syntaktisk sukker" i språket, og sitter igjen med det som er den essensielle "meningen" med setningen. Akkurat hvilke elementer som skal inngå i AST'et må presiseres i hvert tilfelle. Under en kompilering bygges vanligvis et AST for det aktuelle programmet.

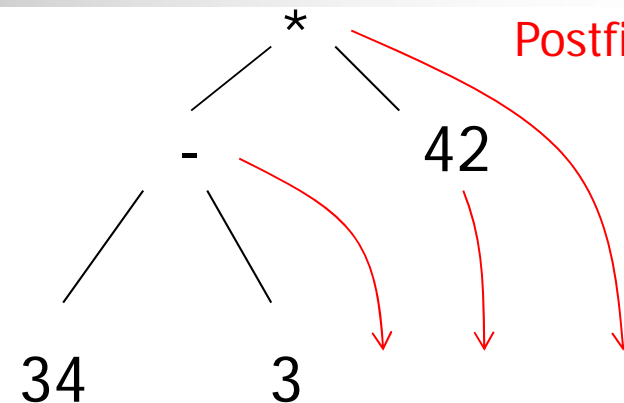


# Abstrakt syntakstre

Prefiks



Postfiks



Man ønsker ofte å skrive ned et tre som en sekvens:

Kan bruke prefiks eller postfiks form (eller infiks eller "omfiks", eller ...)

- Prefiks-form: *først noden, så venstre sub-tre, så høyre sub-tre*):

*\* - 34 3 42* "bruk strykejern fra høyre"

eller som i boka: *OpExp(Times, OpExp(Minus, Const(34), Const(3)), Const(42))*

- Postfiks form: *34 3 - 42 \** "Bruk strykejern fra venstre"

- Infiks form: *34 - 3 \* 42* "Projiser alt rett ned"

**Merk:** Om det er *et kjent antall operander* for hver operator så vil:

Postfiks og prefiks gi **et entydig tre**

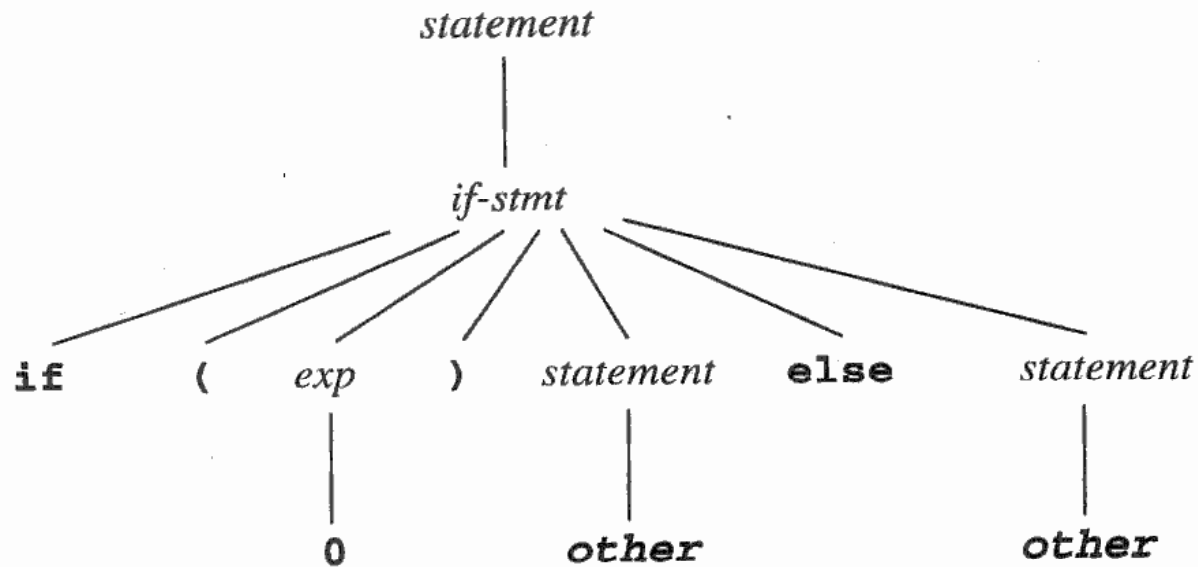
**Men:** Infiks **gir ikke** et entydig tre (trenger i så fall parenteser)

## Flere parserings-trær (konkrete syntaks-trær)

*G1:*

$$\begin{aligned} \text{statement} &\rightarrow \text{if-stmt} \mid \mathbf{other} \\ \text{if-stmt} &\rightarrow \mathbf{if} ( \text{exp} ) \text{statement} \\ &\quad \mid \mathbf{if} ( \text{exp} ) \text{statement} \mathbf{else} \text{statement} \\ \text{exp} &\rightarrow \mathbf{0} \mid \mathbf{1} \end{aligned}$$

*Setning:* **if (0) other else other**





# En annen grammatikk G2 for if-setninger

G2:

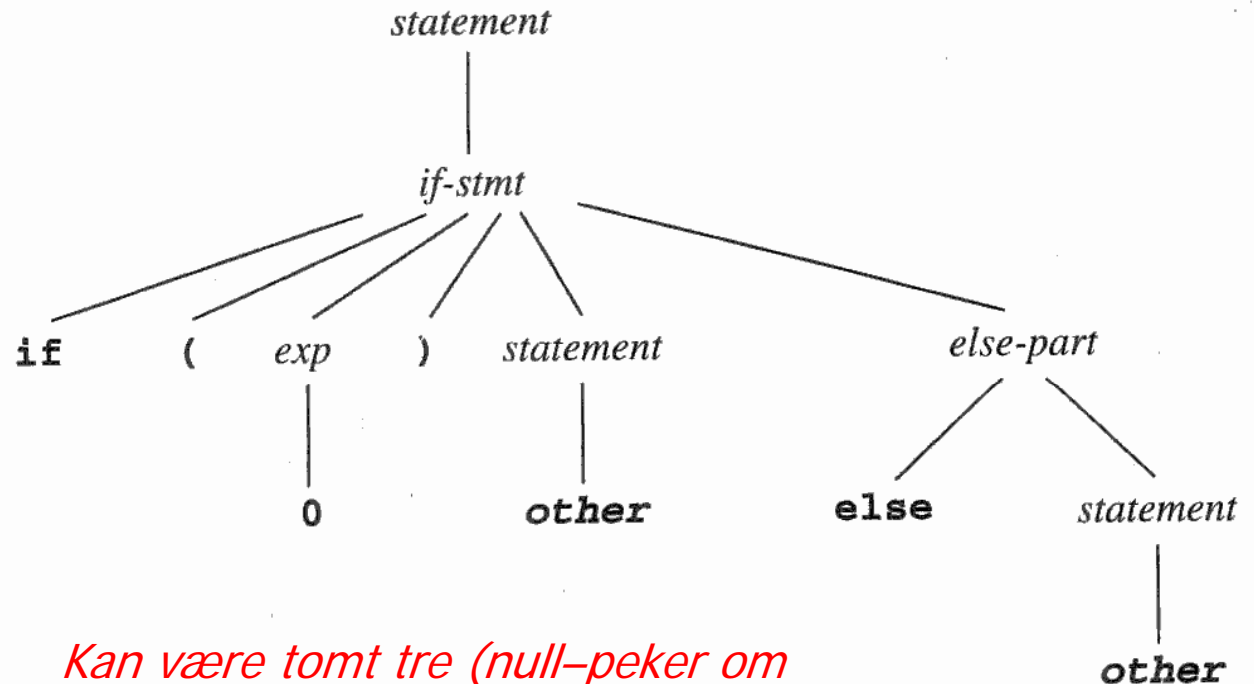
$statement \rightarrow if\text{-stmt} \mid \mathbf{other}$

$if\text{-stmt} \rightarrow \mathbf{if} ( exp ) statement \text{ else-part}$

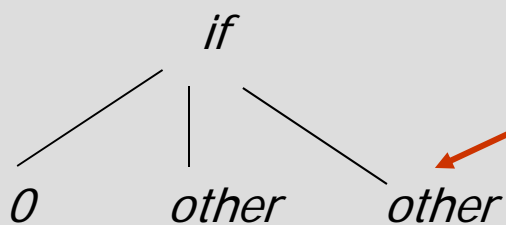
$else\text{-part} \rightarrow \mathbf{else} statement \mid \epsilon$

$exp \rightarrow \mathbf{0} \mid \mathbf{1}$

Den tomme streng angis slik



Abstrakt syntaks-tre for G2 (og også ofte for G1):



Kan være tomt tre (null-peker om det er implementert i Java)

# Flertydige grammatikker

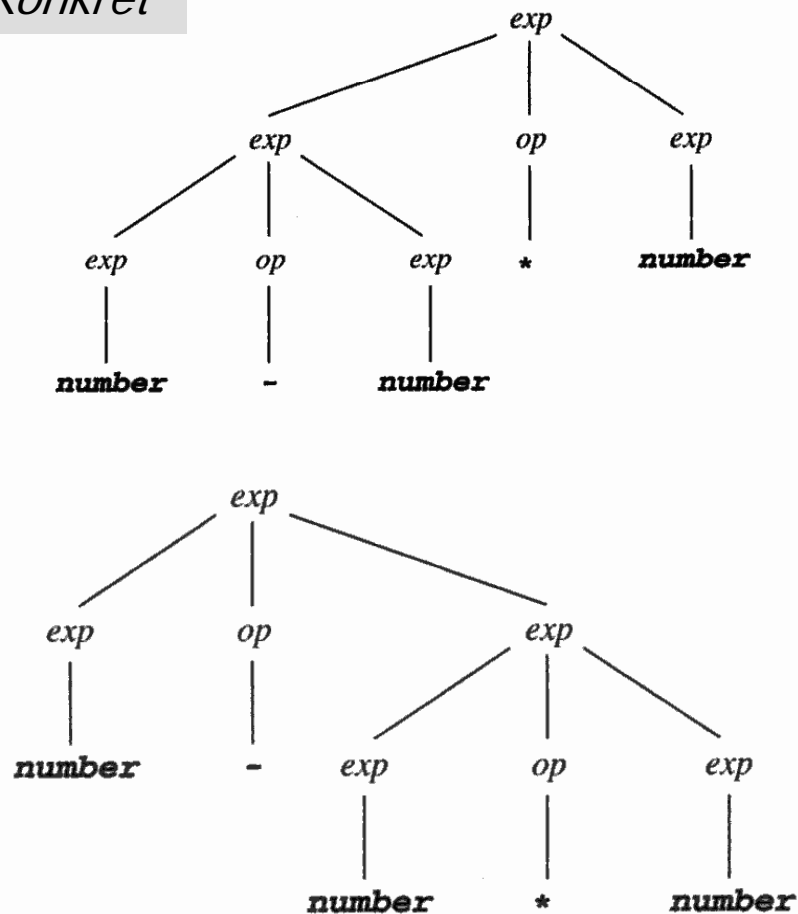
## Analyse av setningen: $n - n * n$

G er *flertydig* hvis det finnes en setning i  $L(G)$  som kan gis **flere** parserings-trær

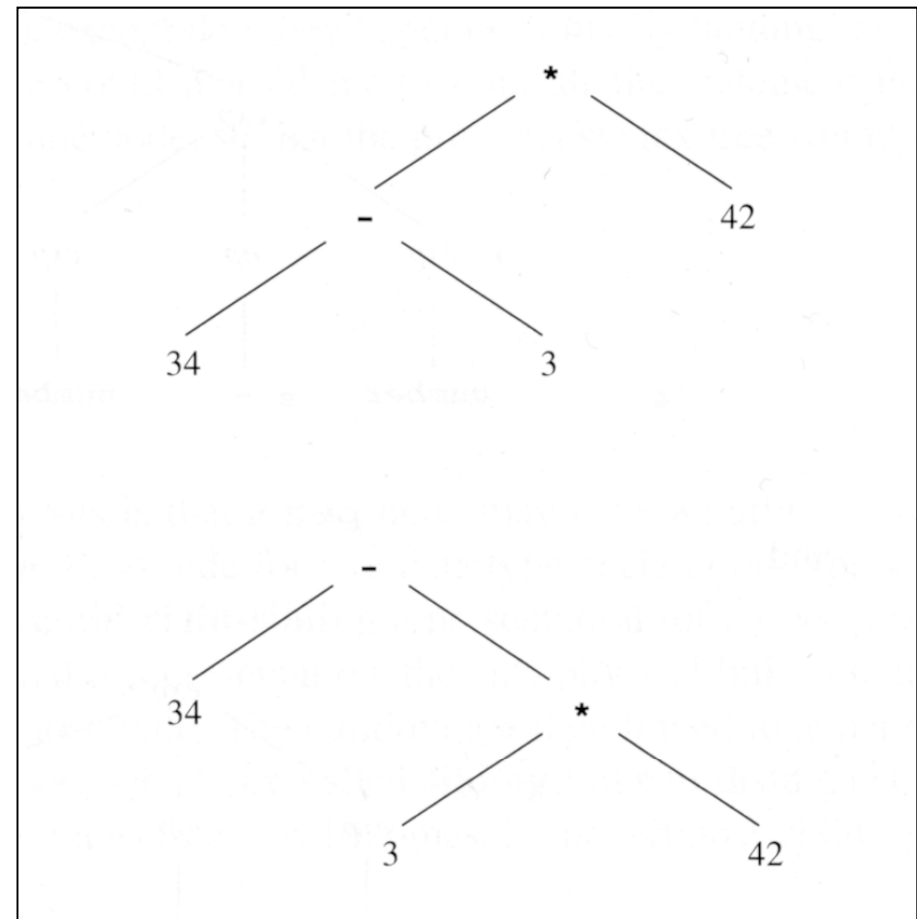
De to trærne kan ofte angi *helt* forskjellige betydninger (eller her: beregninger).

$$\begin{aligned} \text{exp} &\rightarrow \text{exp op exp} \mid (\text{exp}) \mid \mathbf{number} \\ \text{op} &\rightarrow + \mid - \mid * \end{aligned}$$

Konkret



Abstrakt





# Bruk av presedens og assosiativitet for å gjøre flertydige grammatikker endtydige.

- Angir språket ved flertydige grammatikk som

$$\begin{aligned} \text{exp} &\rightarrow \text{exp op exp} \mid ( \text{exp} ) \mid \mathbf{number} \\ \text{op} &\rightarrow + \mid - \mid * \end{aligned}$$

- Oppgir regler for presedens og assosiativitet for hver operasjon, slik at alle setninger får *ett* entydig syntakstre:

+ , -    lav,        venstre ass.  
\* , /    høyere ,    venstre-ass.  
↑ ,      høyerst,    høyre ass.

$$3 + 5 / 3 * 2 + 4 \uparrow 2 \uparrow 3$$

**Betyr:**  $(3 + ((5 / 3) * 2)) + (4 \uparrow (2 \uparrow 3))$

- Dette er helt greit for binære innfiks-operatorer, men fungerer "vanligvis" også greit for *unære* postfiks eller prefiks operatorer



## Om å gjøre flertydige grammatikker entydige, *uten* å bruke presedens og assosiativitet .

$$\begin{aligned} \text{exp} &\rightarrow \text{exp op exp} \mid (\text{exp}) \mid \mathbf{number} \\ \text{op} &\rightarrow + \mid - \mid * \end{aligned}$$

- Presedens for operatorer
  - Noen operasjoner skal gjøres før andre (\* før +)
  - **Ordnes ved** en ekstra ikke-terminal (og "operasjonssett") for hvert presedensnivå (term {+, -}, factor {\*}, ...) – se grammatikken under
- Assosiativitet i grammatikken for operatorer:
  - **Venstre-assosiativitet** Ordnes ved at regler med slike operatorer gjøres "venstre-rekursive" :
$$\text{exp} \rightarrow \text{exp addop term} \mid \text{term}$$
  - **Høyre-assosiativitet** Ordnes tilsvarende, men omvendt.
  - **Ingen assosiativitet** ordnes slik:
$$\text{exp} \rightarrow \text{term addop term} \mid \text{term}$$

Om vi bare har +, -  
og \* (som alle er  
venstreassosiative)  
får vi følgende:

$$\begin{aligned} \text{exp} &\rightarrow \text{exp addop term} \mid \text{term} \\ \text{addop} &\rightarrow + \mid - \\ \text{term} &\rightarrow \text{term mulop factor} \mid \text{factor} \\ \text{mulop} &\rightarrow * \\ \text{factor} &\rightarrow (\text{exp}) \mid \mathbf{number} \end{aligned}$$

# Entydige grammatikker

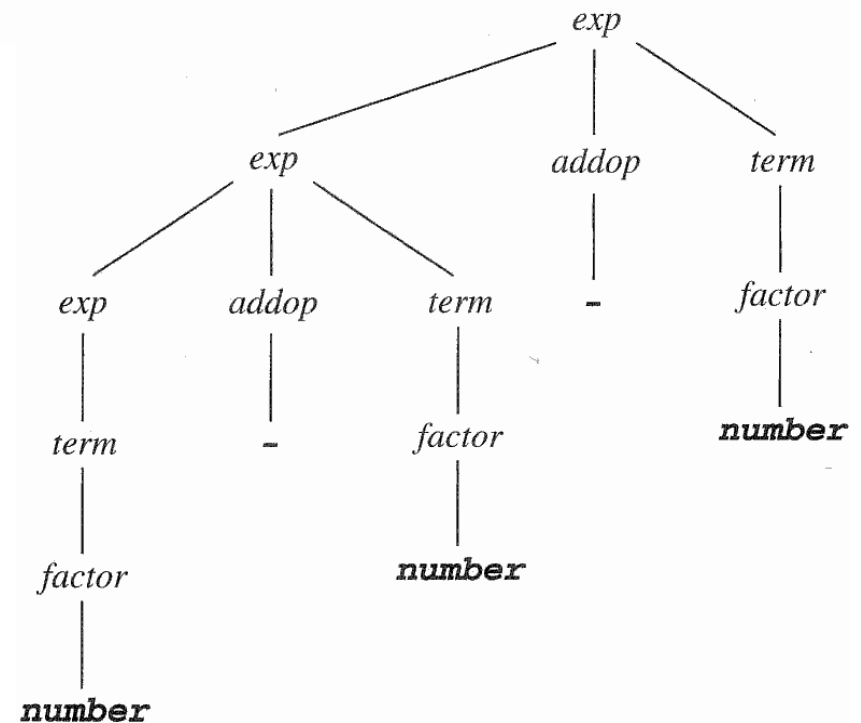
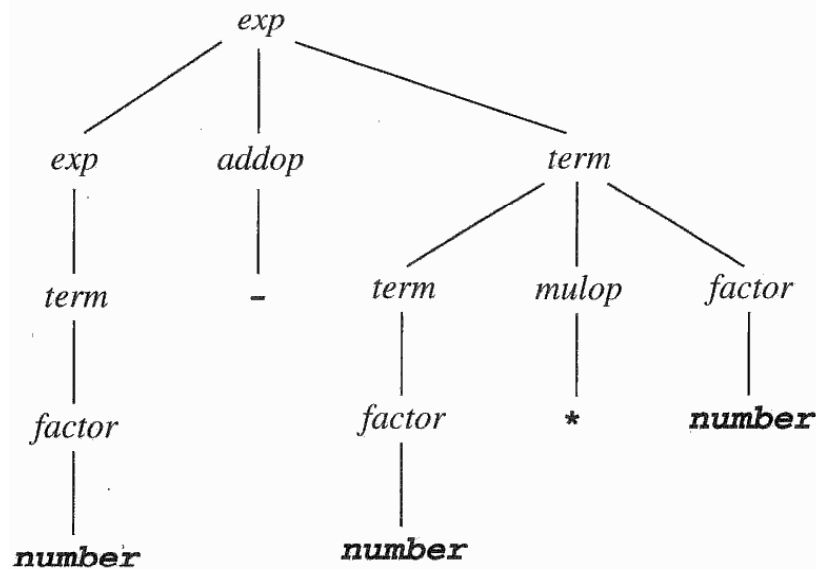
En ekstra ikke-terminal for hvert presedens-nivå

**34-3\*42**

$exp \rightarrow exp \text{ addop } term \mid term$   
 $addop \rightarrow + \mid -$   
 $term \rightarrow term \text{ mulop } factor \mid factor$   
 $mulop \rightarrow *$   
 $factor \rightarrow ( exp ) \mid \mathbf{number}$

Rekkefølgen angir venstre/høyre assosiativitet

**34-3-42**



**Merk:** Spørsmålet om en grammatikk er entydig er generelt "uavgjørbart" 21

## Presendens og assosiativitet i Java

### Operator Precedence

Venstre assosiativ

Java performs operations assuming the following ordering (or *precedence*) rules if parentheses are not used to determine the order of evaluation (operators on the same line are evaluated in left-to-right order subject to the conditional evaluation rule for `&&` and `||`). The operations are listed below from highest to lowest precedence (we use `<exp>` to denote an atomic or parenthesized expression):

postfix ops	<code>[] . (&lt;exp&gt;) &lt;exp&gt; ++ &lt;exp&gt; --</code>
prefix ops	<code>++&lt;exp&gt; --&lt;exp&gt; -&lt;exp&gt; ~&lt;exp&gt; !&lt;exp&gt;</code>
creation/cast	<code>new ((&lt;type&gt;))&lt;exp&gt;</code>
mult./div.	<code>* / %</code>
add./subt.	<code>÷ -</code>
shift	<code>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</code>
comparison	<code>&lt; &lt;= &gt; &gt;= instanceof</code>
equality	<code>== !=</code>
bitwise-and	<code>&amp;</code>
bitwise-xor	<code>^</code>
bitwise-or	<code> </code>
and	<code>&amp;&amp;</code>
or	<code>  </code>
conditional	<code>(&lt;bool_exp&gt;? &lt;&gt;true_val&gt;: &lt;&gt;false_val&gt;)</code>
assignment	<code>=</code>
op assignment	<code>+= -= *= /= %=</code>
bitwise assign.	<code>&gt;&gt;= &lt;&lt;= &gt;&gt;&gt;=</code>
boolean assign.	<code>&amp;= ^=  =</code>

Ikke-essensiell  
flertydighet

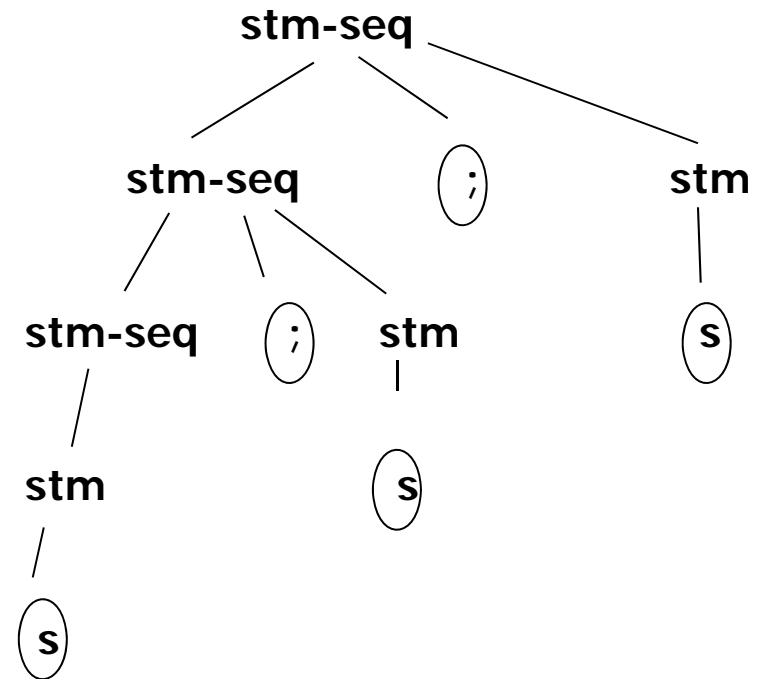
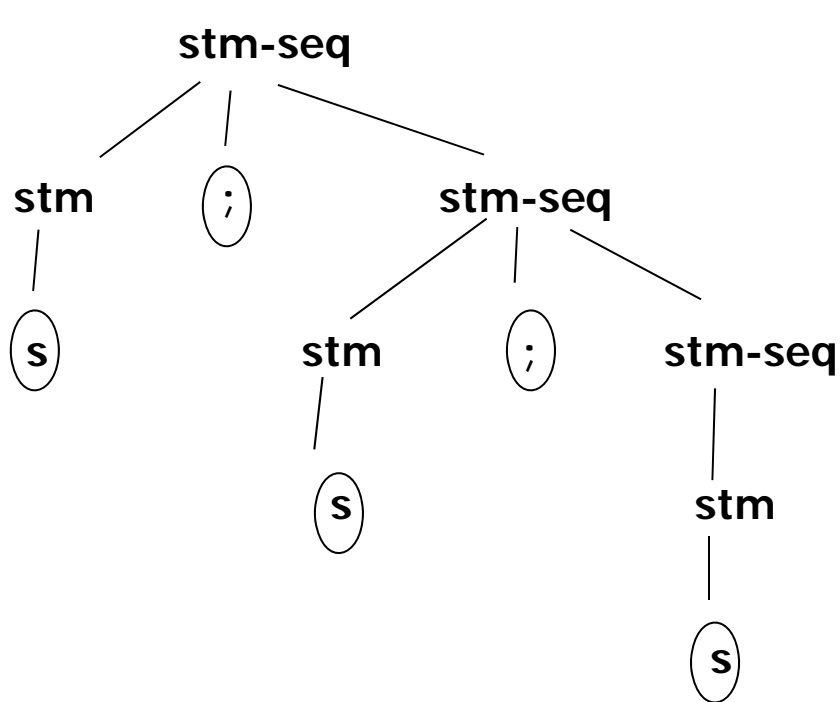
$stm\text{-seq} \rightarrow stm\text{-seq}; stm \mid stm$

$stm\text{-seq} \rightarrow stm; stm\text{-seq} \mid stm$

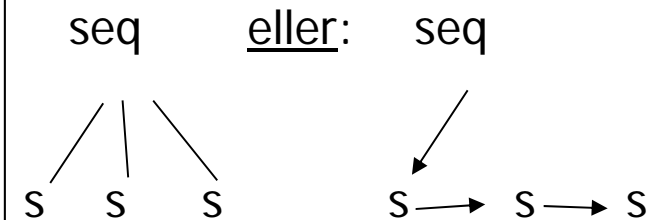
$stm \rightarrow s$

venstre-assos.

høyre-assos.



Kan like gjerne representeres  
slik i praksis:



## "Dangelig else" - problemet

- Problem: Hvilken **if**-setning skal vi koble **else** til?

Slik: ( )

```
if (0) if (1) other else other
```

eller slik: ( )

- Grammatikken under er flertydig, se neste foil:

$$\text{statement} \rightarrow \text{if-stmt} \mid \mathbf{other}$$
$$\text{if-stmt} \rightarrow \mathbf{if} ( \text{exp} ) \text{ statement}$$
$$\quad \quad \quad \mid \mathbf{if} ( \text{exp} ) \text{ statement} \mathbf{else} \text{ statement}$$
$$\text{exp} \rightarrow \mathbf{0} \mid \mathbf{1}$$

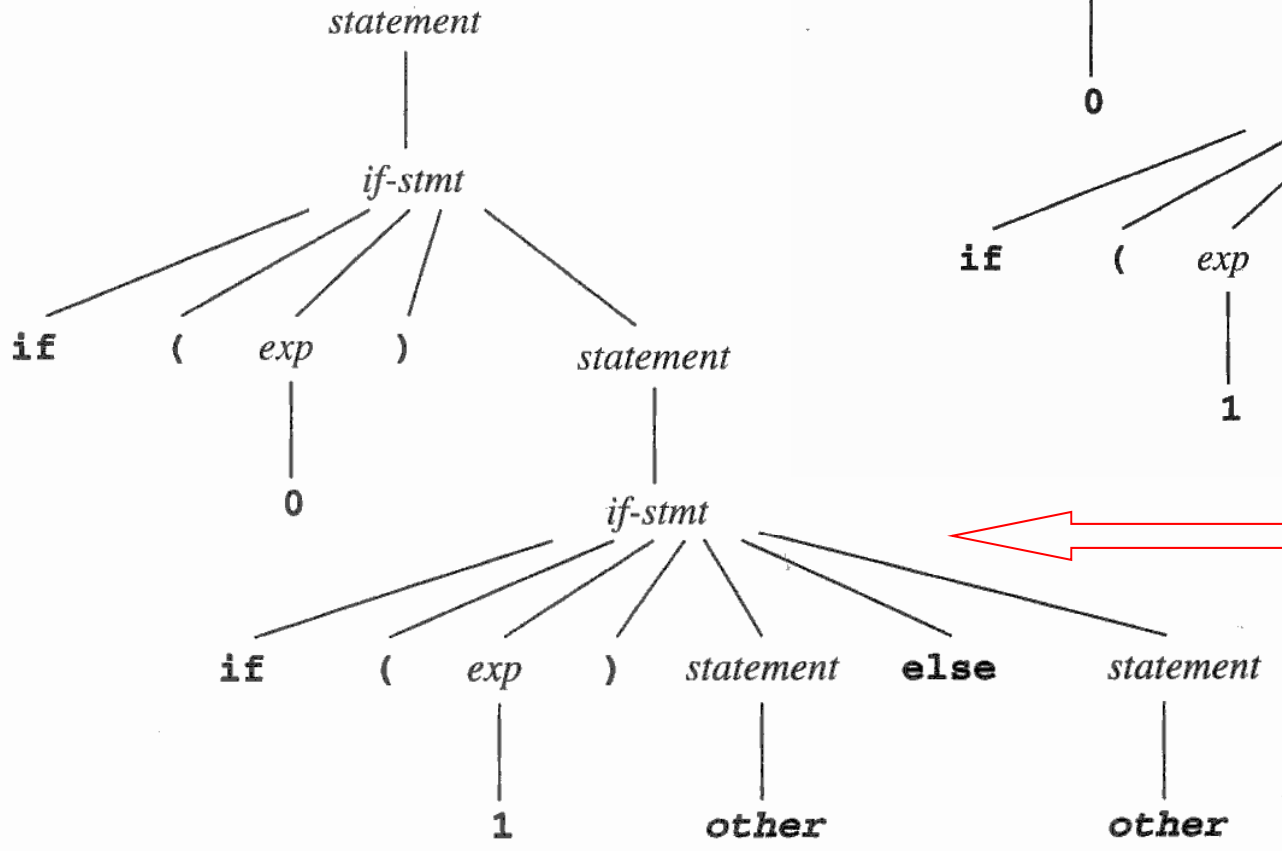
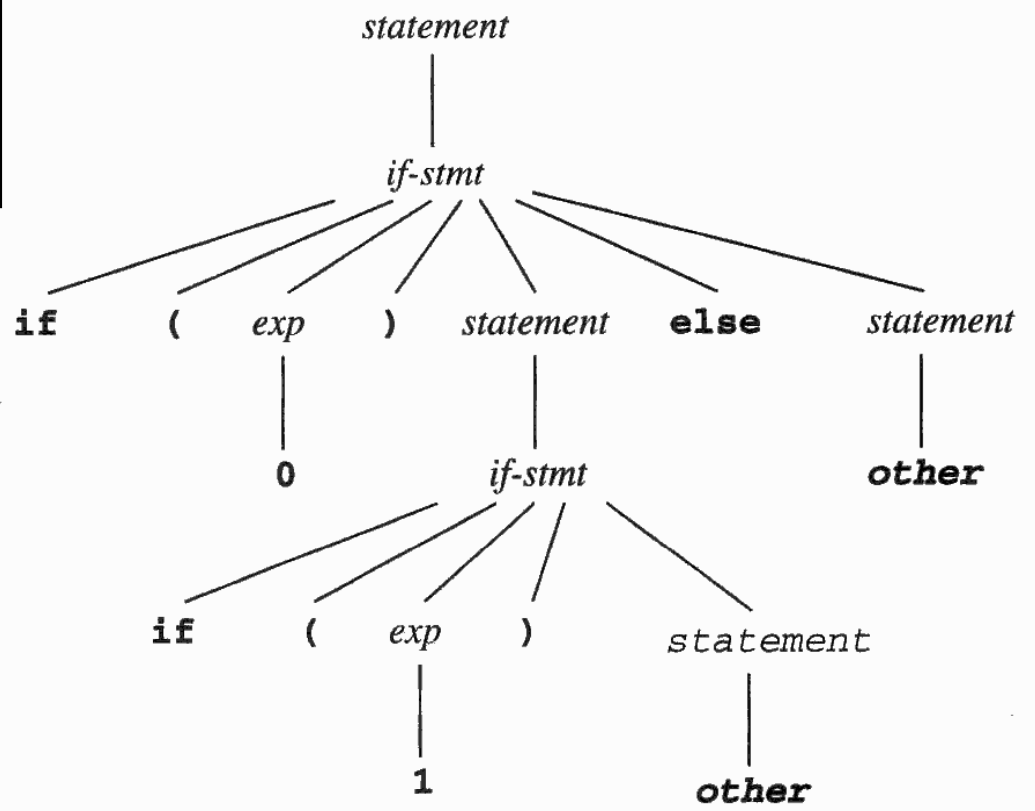


# To træer for samme sætning

$statement \rightarrow if-stmt \mid other$   
 $if-stmt \rightarrow if ( exp ) statement$   
 $\quad \quad \quad \mid if ( exp ) statement else statement$   
 $exp \rightarrow 0 \mid 1$

*Setning:*

`if (0) if (1) other else other`



Vanlig regel er denne:  
 La else bli koblet til nærmeste "ledige" if

# Eks: Entydig grammatikk for if-setning.

## Gir "vanlig" løsning

`if (0) if (1) other else other`

$statement \rightarrow matched-stmt \mid unmatched-stmt$   
 $matched-stmt \rightarrow \mathbf{if} ( exp ) matched-stmt \mathbf{else} matched-stmt \mid \mathbf{other}$   
 $unmatched-stmt \rightarrow \mathbf{if} ( exp ) statement$   
 $\quad \mid \mathbf{if} ( exp ) matched-stmt \mathbf{else} unmatched-stmt$   
 $exp \rightarrow 0 \mid 1$

Idé:

Kan ikke ha en unmatched inne i en matchet

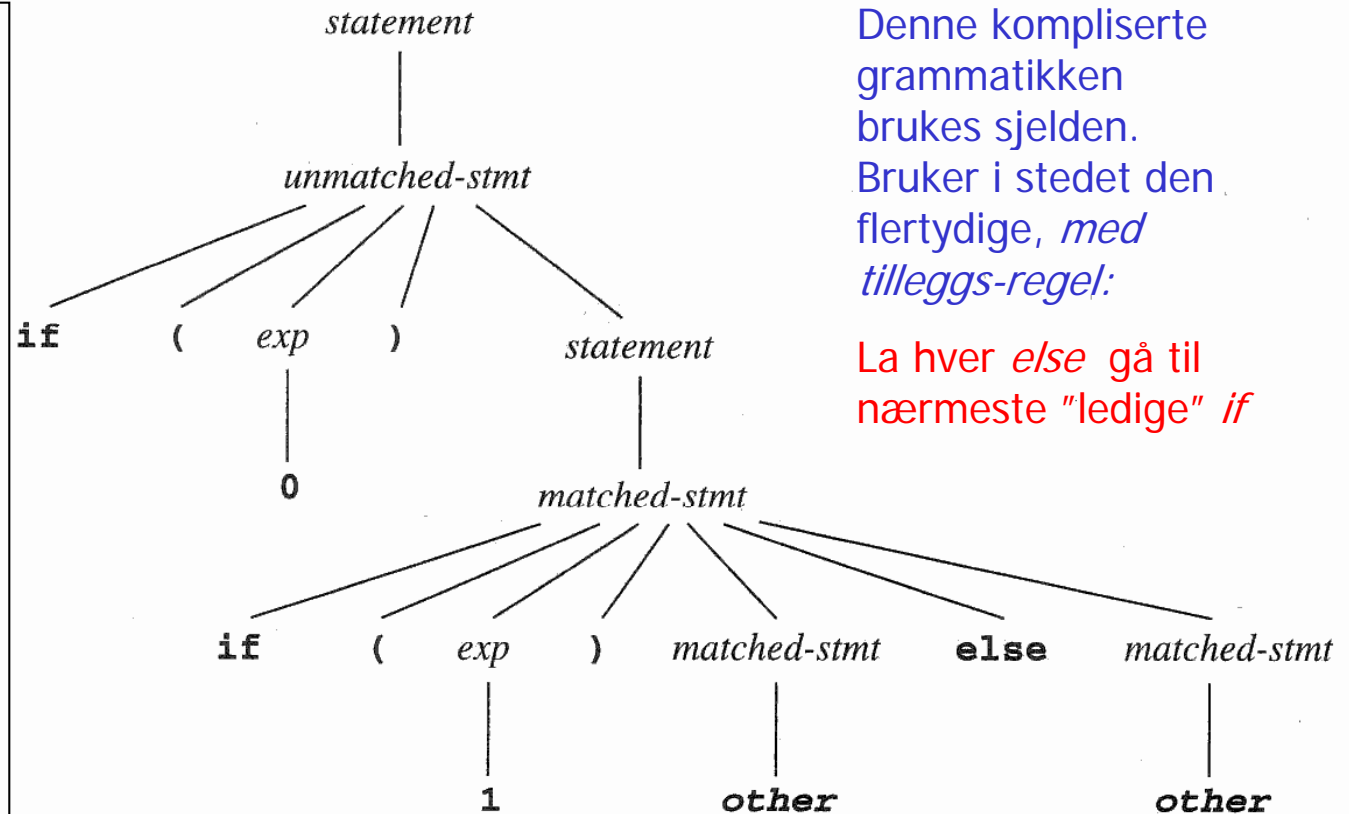
**matched-stmt:**

Inneholder selv en else og kan derfor **ikke** kobles med etterfølgende else

**unmatched-stmt:**

Har ingen else og kan kobles med etterfølgende else

**Spørsmål:** Er det sikkert at denne kan generere alle "lovlige" setninger (de fra den kortere flertydige grammatikken på forrige to foiler)?



Denne kompliserte grammatikken brukes sjelden.

Bruker i stedet den flertydige, med tilleggs-regel:

La hver *else* gå til nærmeste "ledige" *if*

# Utvidet BNF (EBNF)

Idé: Man kan generelt bruke "regulære uttrykk" på høyresiden i produksjoner

Vanlig:  $\alpha^*$  skrives:  $\{\alpha\}$   $\alpha$  er en streng av terminaler og ikke-terminaler  
 $\alpha?$  skrives:  $[\alpha]$

Eksempel:

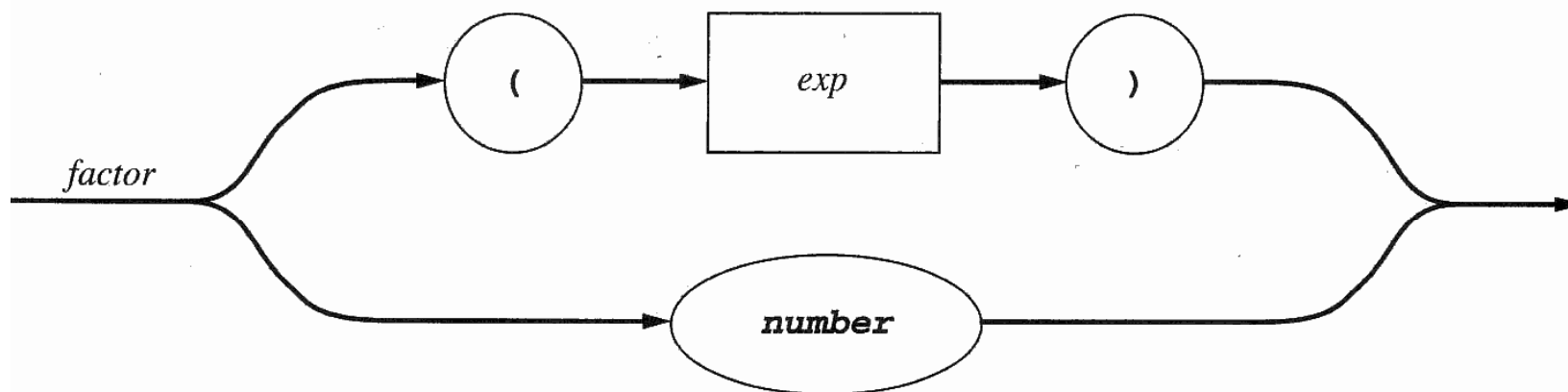
$exp \rightarrow exp ( "+" | "-" | "*" ) exp | "( exp )" | \mathbf{number}$

Meta-symbol                      ikke-meta

$A \rightarrow A \alpha | \beta$  kan skrives:  $A \rightarrow \beta \{\alpha\}$   
 $A \rightarrow \alpha A | \beta$  kan skrives:  $A \rightarrow \{\alpha\} \beta$   
 $stm\text{-}seq \rightarrow stm \{ ; stm \}$  eller  $stm\text{-}seq \rightarrow \{ stm ; \} stm$   
 $if\text{-}setn \rightarrow \underline{if} (expr) stm [ \underline{else} stmt ]$

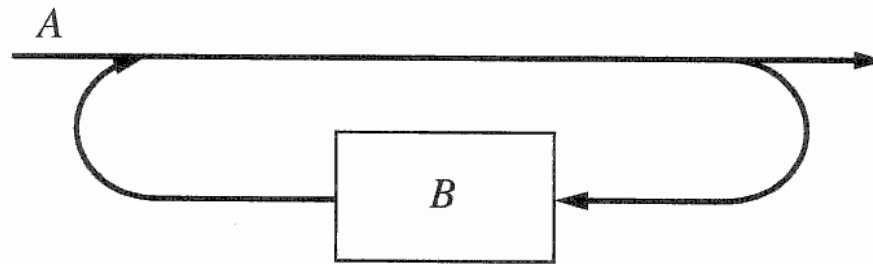
Merk: For en del autom. verktøy *må* man bruke basal BNF.

# Syntaks-diagrammer

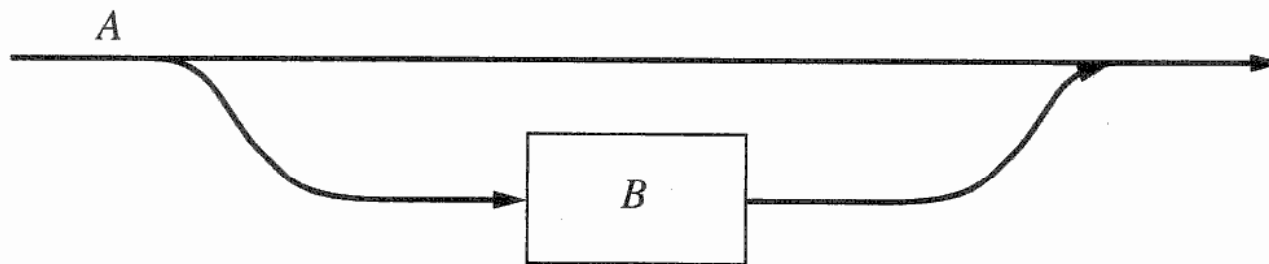
$$factor \rightarrow ( exp ) \mid number$$


Merk: "Entydige syntakstre" og andre liknende begreper er ikke så naturlig å definere her

$A \rightarrow \{ B \}$



$A \rightarrow [ B ]$



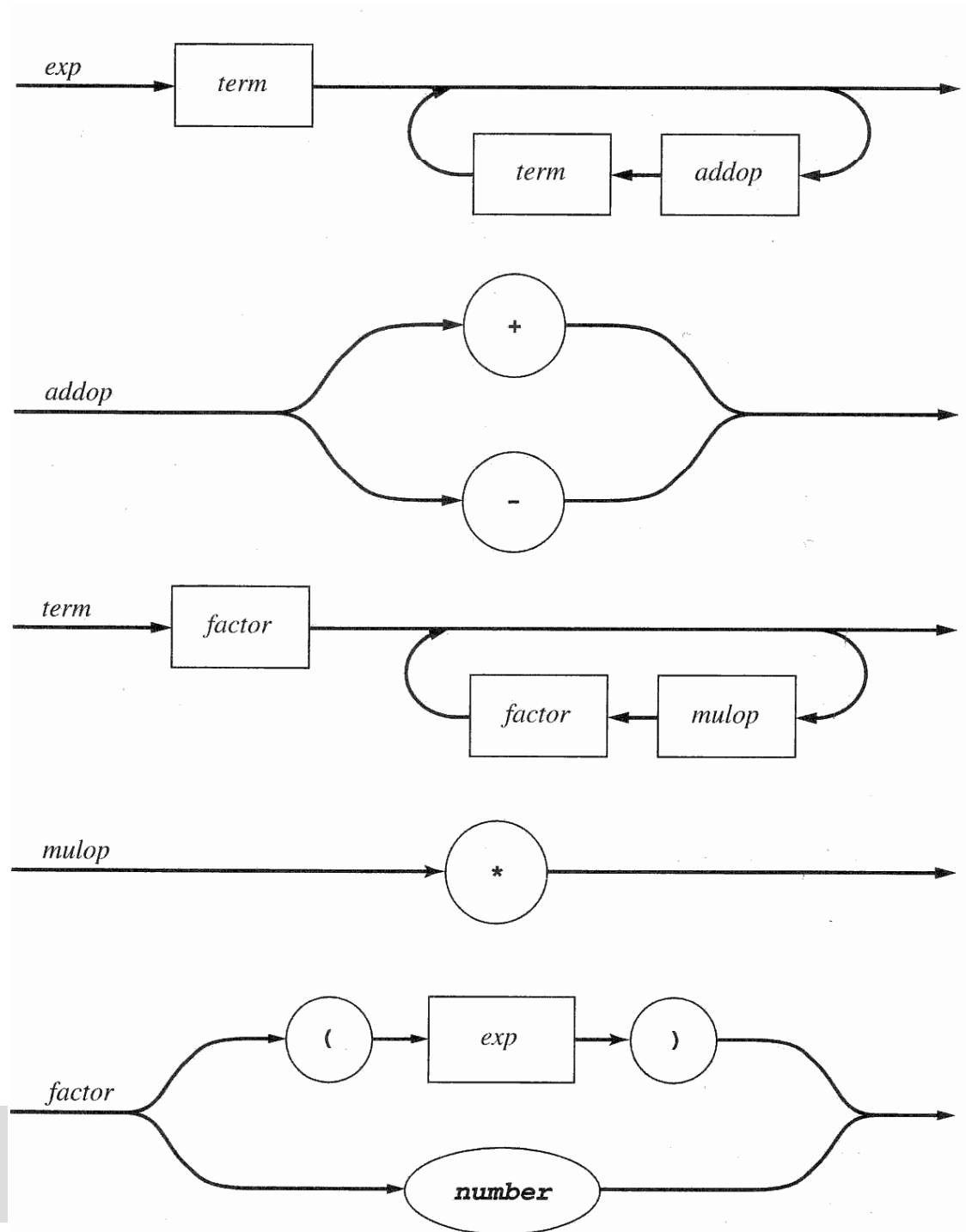
## BNF-grammatikk:

$exp \rightarrow exp \text{ addop } term \mid term$   
 $addop \rightarrow + \mid -$   
 $term \rightarrow term \text{ mulop } factor \mid factor$   
 $mulop \rightarrow *$   
 $factor \rightarrow ( exp ) \mid \mathbf{number}$

EBNF-grammatikk for samme "språket". De tilsvarer mer direkte syntaksdiagrammene:

$exp \rightarrow term \{ addop term \}$   
 $addop \rightarrow + \mid -$   
 $term \rightarrow factor \{ mulop factor \}$   
 $mulop \rightarrow *$   
 $factor \rightarrow ( exp ) \mid \mathbf{number}$

**Merk:** Her må *assosiativitet* (men ikke *presedens*) gis i tillegg





# Chomsky-hierarkiet

$a$  er et vilkårlig terminalsymbol  
 $\beta, \alpha, \gamma$  er vilkårlig streng av terminal- og ikke-terminalsymboler  
 $A, B$  er ikke-terminaler

- Type 0 – språk
  - Urestriktede prod.:  $\alpha \rightarrow \beta, \quad \alpha \neq \varepsilon$  ( $\alpha$  er ikke-tom)
- Type 1 – språk
  - Kontekst-sensitive produksjoner:  $\beta A \gamma \rightarrow \beta \alpha \gamma$
- **Type 2 – språk** (det vi bruker til vanlig syntaks-beskrivelse)
  - Kontkstfrie prod., (E)BNF:  $A \rightarrow \alpha$
- **Type 3 – språk** (det vi bruker for leksemer i "skanneren")
  - Regulære språk:
    - Regulære uttrykk
    - NFA
    - DFA

Produksjoner bare på formen:

$A \rightarrow B a$  og  $A \rightarrow a$

eller bare på formen:

$A \rightarrow a B$  og  $A \rightarrow a$



## Hvorfor ikke bare ha én (stor!) grammatikk som sier alt om språket: **leksemer, form og semantikk** ?

---

- Kunne vi ikke laget én (stor) grammatikk som sa 'alt' om språket??
  - F.eks. det som tas av skanneren: hvordan tall, variable etc. er definert
  - Som også sa at det skal være samme type på hver side av en tilordning
- Vi gjør ikke dette fordi:
  - En slik grammatikk ville i det minste bli 'uhåndterlig stor'
  - Faktisk umulig å formulere visse aspekter ved programmeringsspråk ved den type (kontekst-frie) grammatikker vi bruker
    - F.eks. at alle variable er deklarerert
  - Mye greiere å ta:
    - enkle ting i skanneren (der regulære grammatikker passer)
    - setningsformen i parseren (der kontekstfrie grammatikker passer)
    - mer kompliserte krav i semantikk-sjekkeren (skrives som et program)
      - jfr. **samlebåndsproduksjon av biler (bilen lages i flere steg)**
- Må ofte jobbe med hvordan vi formulerer en grammatikk for at den skal gi en god/riktig parser
  - flere måter å formulere grammatikken for et språk



# BNF-grammatikk for TINY

*program* → *stmt-sequence*  
*stmt-sequence* → *stmt-sequence* ; *statement* | *statement*  
*statement* → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*  
*if-stmt* → **if** *exp* **then** *stmt-sequence* **end**  
          | **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**  
*repeat-stmt* → **repeat** *stmt-sequence* **until** *exp*  
*assign-stmt* → **identifier** := *exp*  
*read-stmt* → **read** **identifier**  
*write-stmt* → **write** *exp*  
*exp* → *simple-exp* *comparison-op* *simple-exp* | *simple-exp*  
*comparison-op* → < | =  
*simple-exp* → *simple-exp* *addop* *term* | *term*  
*addop* → + | -  
*term* → *term* *mulop* *factor* | *factor*  
*mulop* → \* | /  
*factor* → ( *exp* ) | **number** | **identifier**



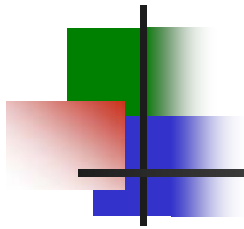
## Nodestruktur i C for syntakstrær til TINY

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK}
           StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* ExpType is used for type checking */
typedef enum {Void,Integer,Boolean} ExpType;


#define MAXCHILDREN 3

typedef struct treeNode
{ struct treeNode * child[MAXCHILDREN];
  struct treeNode * sibling;
  int lineno;
  NodeKind nodekind;
  union { StmtKind stmt; ExpKind exp;} kind;
  union { TokenType op;
         int val;
         char * name; } attr;
  ExpType type; /* for type checking of exps */
} TreeNode;
```



# Nodestruktur i C for Tiny

*If, Repeat, Assign, Read, Write* - tegnes: 

*Op, Const, id* - tegnes: 

*Brukes bare for exp-kind*

*kind:*

*stm-kind / expr-kind*

*attr:*

*op / val / name*

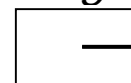
*lineno:*



*type:*



*sibling:*



*child:*



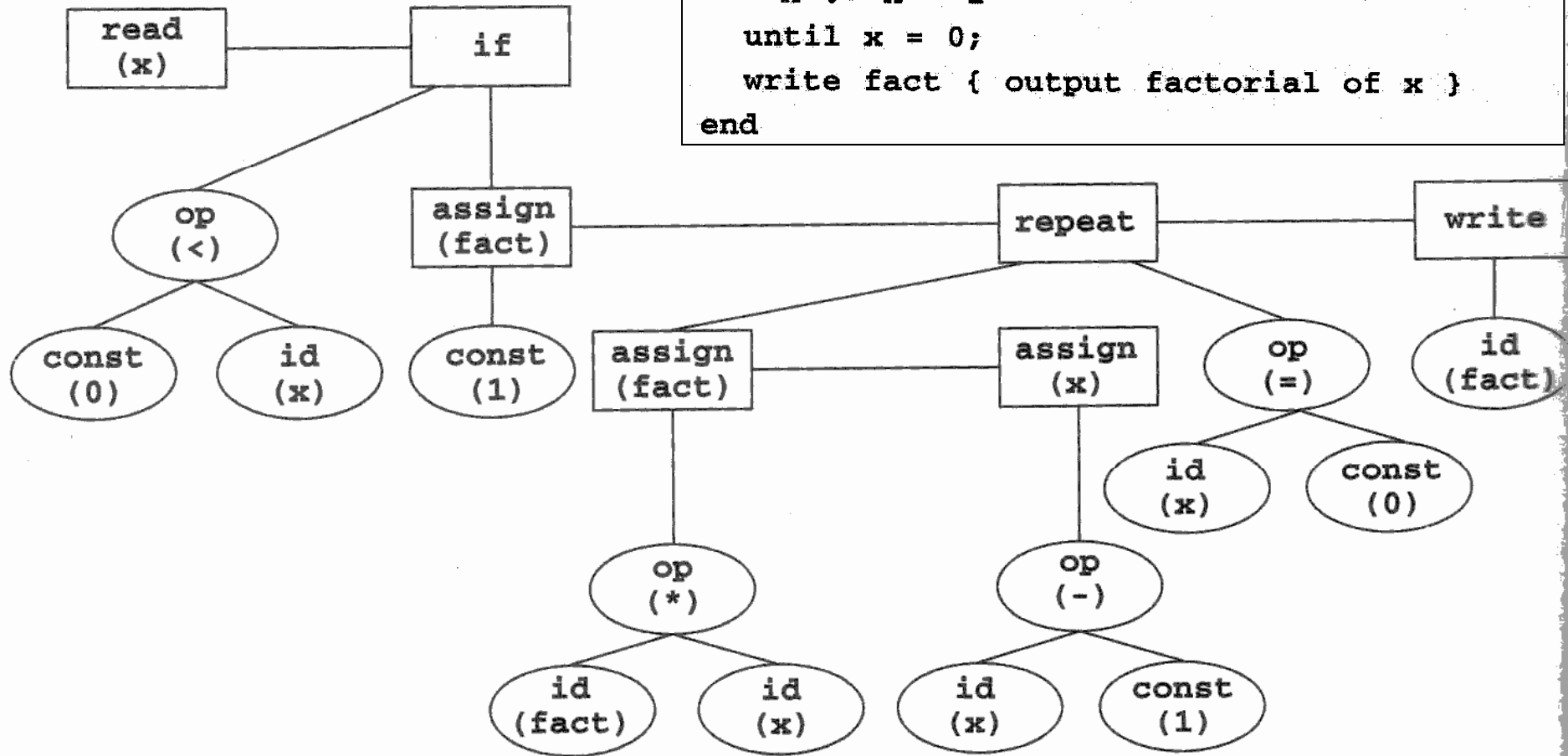
Denne node-strukturen kan uttrykkes bedre om vi bruker et OO-språk som implementasjonsspråk, der vi har klasser og subklasser.

Da får vi et helt hierarki av klasser som beskriver de forskjellige nodetyperne

Dette skal vi gjøre i Obligen!

## Abstrakt syntakstre for et Tiny-program

```
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
write fact { output factorial of x }
end
```



*Spørsmål: Hvordan kunne klasse-hierarkiet se ut som beskriver disse node-typer?*



## Noen spørsmål om Tiny-grammatikken

*program* → *stmt-sequence*  
*stmt-sequence* → *stmt-sequence ; statement | statement*  
*statement* → *if-stmt | repeat-stmt | assign-stmt | read-stmt | write-stmt*  
*if-stmt* → **if** *exp* **then** *stmt-sequence* **end**  
          | **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**  
*repeat-stmt* → **repeat** *stmt-sequence* **until** *exp*  
*assign-stmt* → **identifier** := *exp*  
*read-stmt* → **read identifier**  
*write-stmt* → **write** *exp*  
*exp* → *simple-exp comparison-op simple-exp | simple-exp*  
*comparison-op* → < | =  
*simple-exp* → *simple-exp addop term | term*  
*addop* → + | -  
*term* → *term mulop factor | factor*  
*mulop* → \* | /  
*factor* → ( *exp* ) | **number** | **identifier**

- Er grammatikken entydig?
- Hva om vi vil tillate tomme setninger
- Hva om vi vil ha semikolon etter og ikke mellom setningene?
- Hva slags assosiativitet og presedens er det for operatorene?