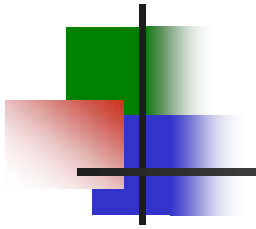
INF 5110, 9. februar 2010

Stein Krogdahl

Dagens Tema: Grammatikker

Kap. 3 i K. C. Louden

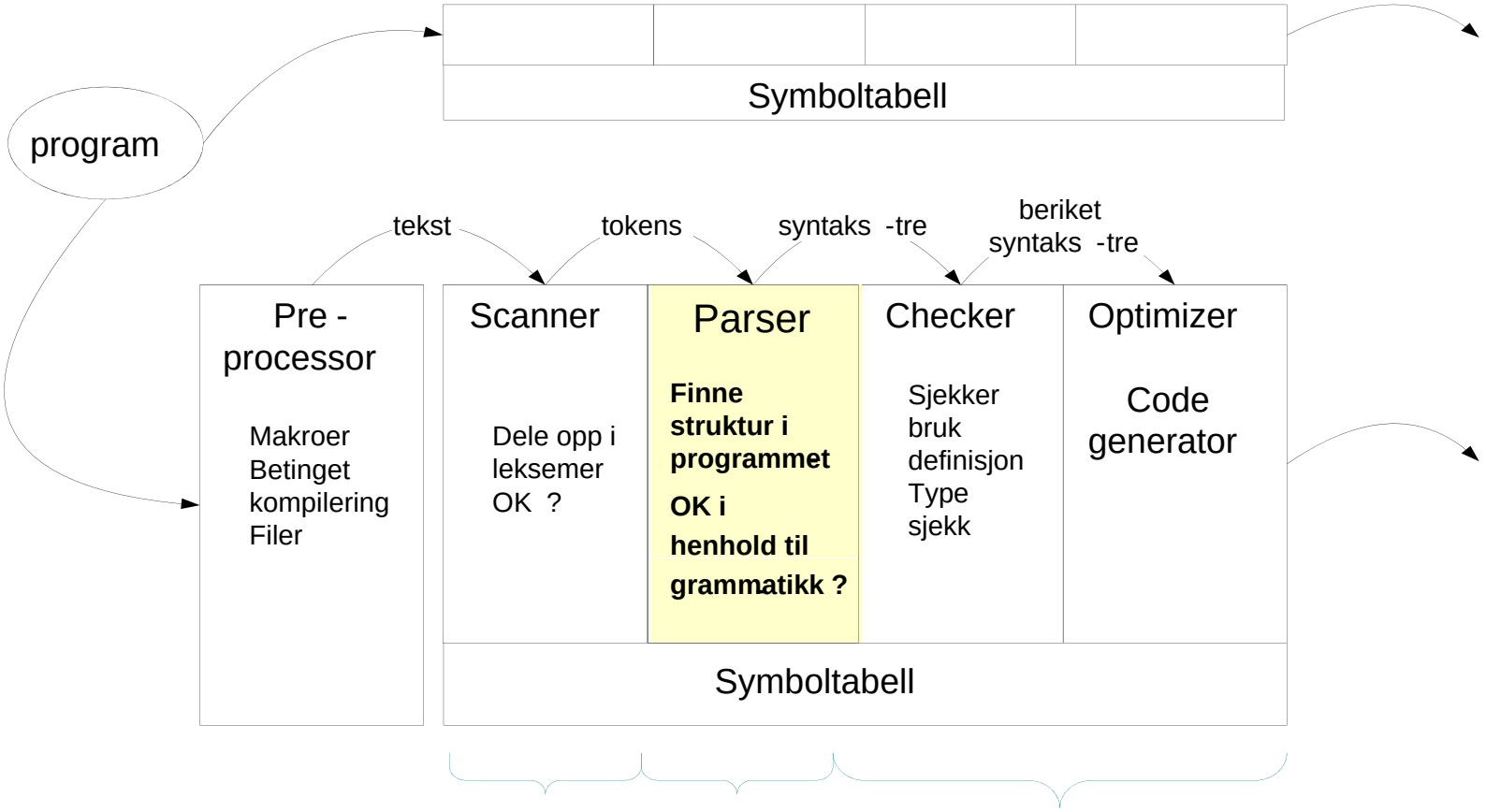
Min Foil-stil: Ofte mer tekst enn man helt kan få med seg på forelesningen, for at de skal være gode til repetisjon



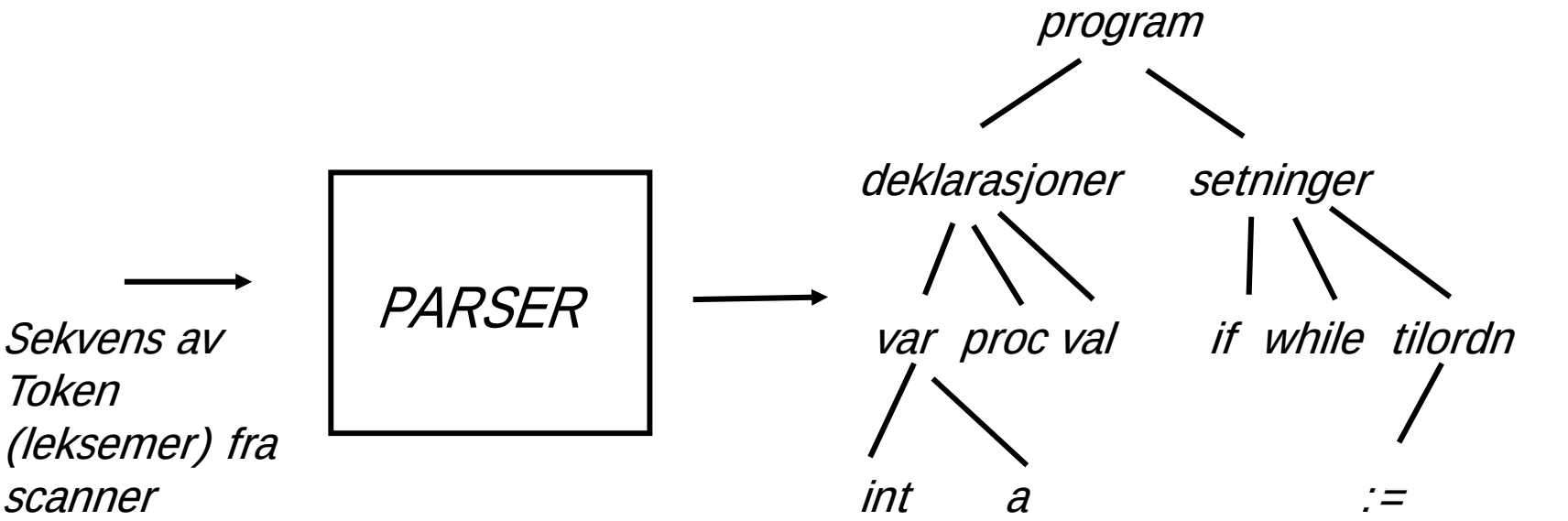
Foreleser: Stein Krogdahl

- 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:
    - Finn beste utplukk som tilfredsstiller bestemte krav o.l.
  - Formell verifikasjon av programmer:
    - Men rettet mot direkte anvendelse
  - Programmeringsspråk: Design og implementasjon
  - Nå: Mest SWAT-prosjektet.

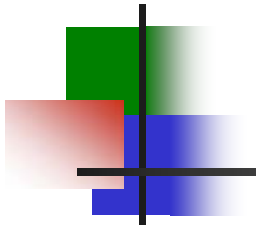
Hvor er vi nå - kap. 3, 4 og 5:



Forenklet skisse av hva en parser gjør



Syntaks-tre:
"Abstrakt" eller "konkret". Dette er typisk abstrakt.

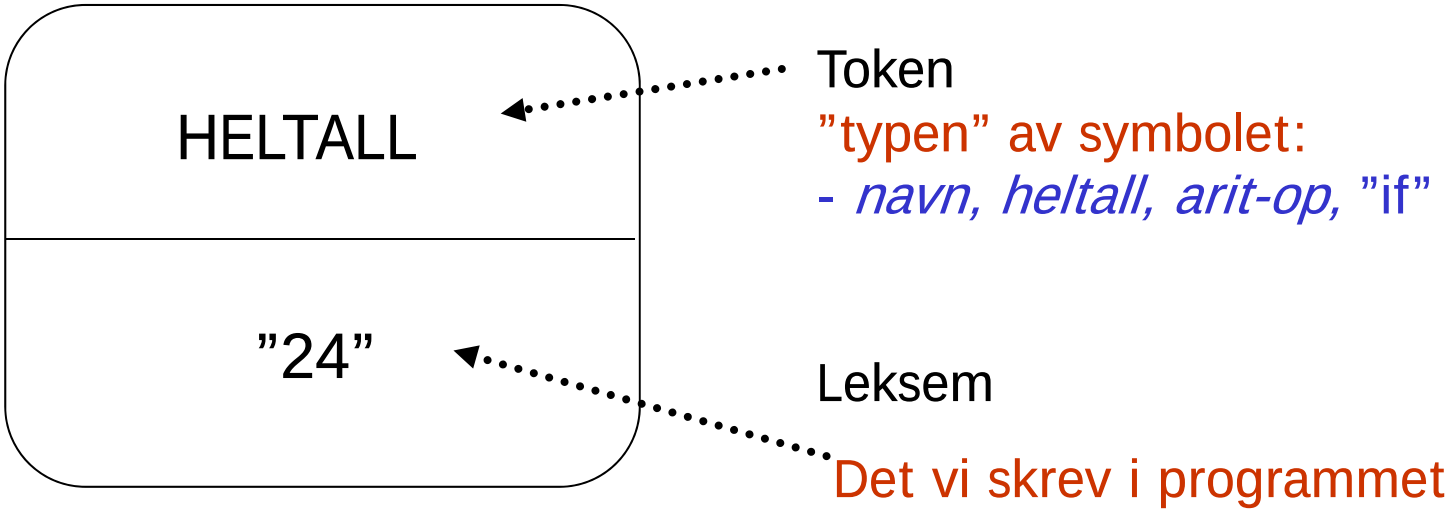


Oversikt – kap 3 (grunnleggende om grammatikker)

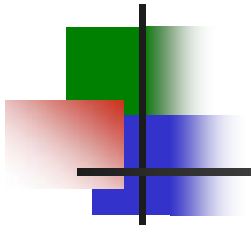
- Hva er en grammatikk?
- Kontekstfrie grammatikker og BNF-notasjon
- Parserings-trær og abstrakte syntaks-trær
- Entydige/flertydige grammatikker
- Utvidet notasjon: EBNF og syntaksdiagrammer
- Eksempel
  - Tiny

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?)



Hva er en grammatikk?

- 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 sammensettinger 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 → exp op exp | **(**exp**)** | **number**

  op → **+** | **-** | *****

  - Metasymboler: **"→"** (leses: "Kan ha formene") ,"|" (leses: "eller")
  - Ikke-terminaler: exp, op
  - Terminaler : **number**, **(**,**)**, *****, **+**, **-**
  - Startsymbol: exp

- En tradisjonell måte (Algol 60 rapporten):

```
<exp> ::= <exp> <op> <exp> | (<exp>) | NUMBER
<op> ::= + | - | *
```

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

exp → *exp* ("**+**" | "**-**" | "*****") *exp* | "**(**" *exp* "**)**" | ***number***

Flere måter å skrive den samme grammatikken

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

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

- 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 **)**
$exp \rightarrow$ **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) $\Rightarrow$ **(number** $op\ exp$ **)** $op\ exp$	$[exp \rightarrow$ **number]**
(5) $\Rightarrow$ **(number** $-\ exp$ **)** $op\ exp$	$[op \rightarrow$ **-]**
(6) $\Rightarrow$ **(number - number)** $op\ exp$	$[exp \rightarrow$ **number]**
(7) $\Rightarrow$ **(number - number) *** exp	$[op \rightarrow$ ***]**
(8) $\Rightarrow$ **(number - number) * number**	$[exp \rightarrow$ **number]**

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

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

grammatikk *streng med bare terminal-symboler*

Avledning (høyreavledning)

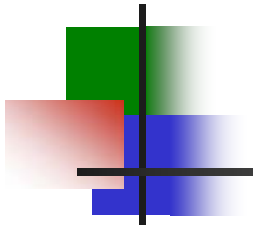
av: (number – number) * number

$exp \rightarrow exp\ op\ exp$
$exp \rightarrow$ **(** exp **)**
$exp \rightarrow$ **number**
$op \rightarrow$ **+**
$op \rightarrow$ **-**
$op \rightarrow$ *****

Startsymbol		Produksjon brukt
(1) exp	$\Rightarrow exp\ op\ exp$	$[exp \rightarrow exp\ op\ exp]$
(2)	$\Rightarrow exp\ op$ **number**	$[exp \rightarrow$ **number]**
(3)	$\Rightarrow exp$ * **number**	$[op \rightarrow$ *]
(4)	$\Rightarrow$ **(** exp **) * number**	$[exp \rightarrow$ **(** exp **)**]
(5)	$\Rightarrow$ **(** $exp\ op\ exp$ **) * number**	$[exp \rightarrow exp\ op\ exp]$
(6)	$\Rightarrow$ **(**$exp\ op$ **number) * number**	$[exp \rightarrow$ **number]**
(7)	$\Rightarrow$ **(**exp **- number) * number**	$[op \rightarrow$ -]
(8)	$\Rightarrow$ **(number - number)*number**	$[exp \rightarrow$ **number]**

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

Det finnes masse andre rekkefølger å avlede en setning fra grammatikken



Opplagte krav til en fornuftig grammatikk

Alle ikke-terminaler:

- Må kunne inngå i en streng avledet fra startsymbolet
- Må kunne avledes videre til noe som bare inneholder terminal-symboler

- Eks:

  $A \rightarrow B\ x$

  $B \rightarrow A\ y$

  $C \rightarrow\ z$

  - C 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)

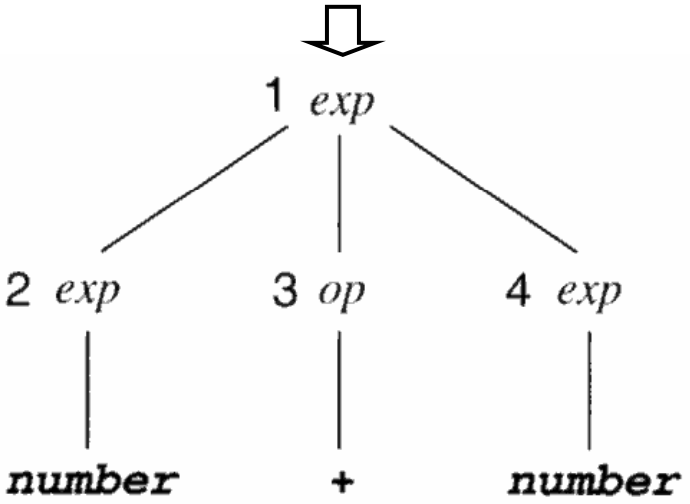
(1) $exp \Rightarrow exp\ op\ exp$

(2) $\Rightarrow$ **number** $op\ exp$

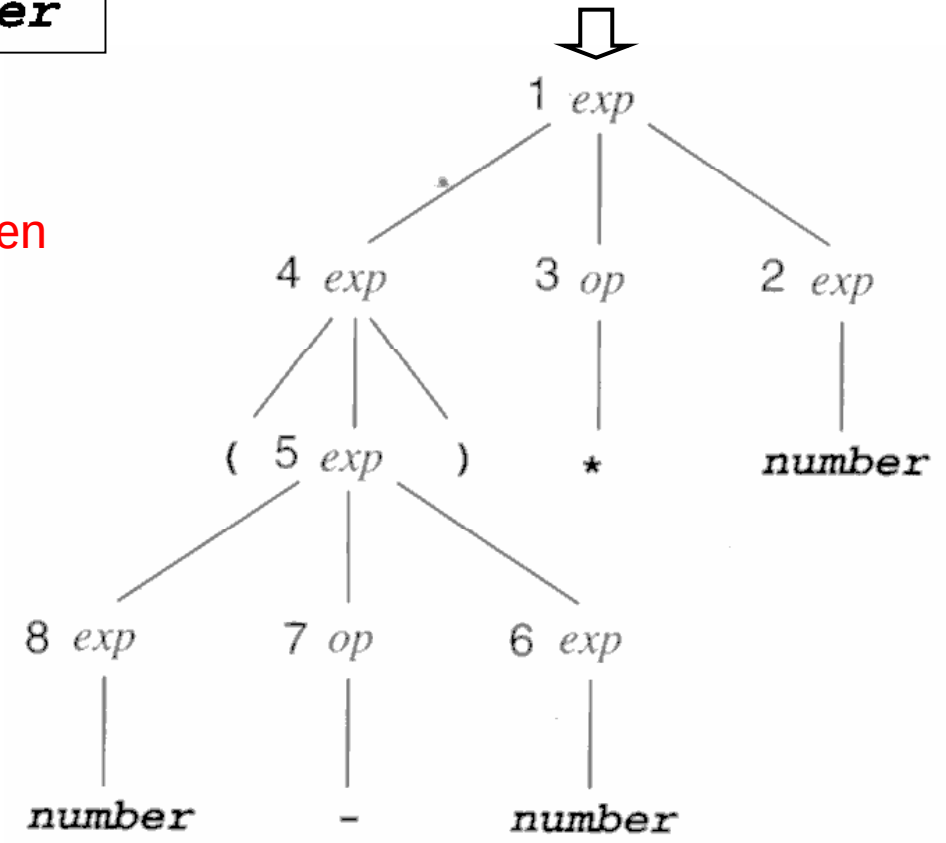
(3) $\Rightarrow$ **number** + exp

(4) $\Rightarrow$ **number** + **number**

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

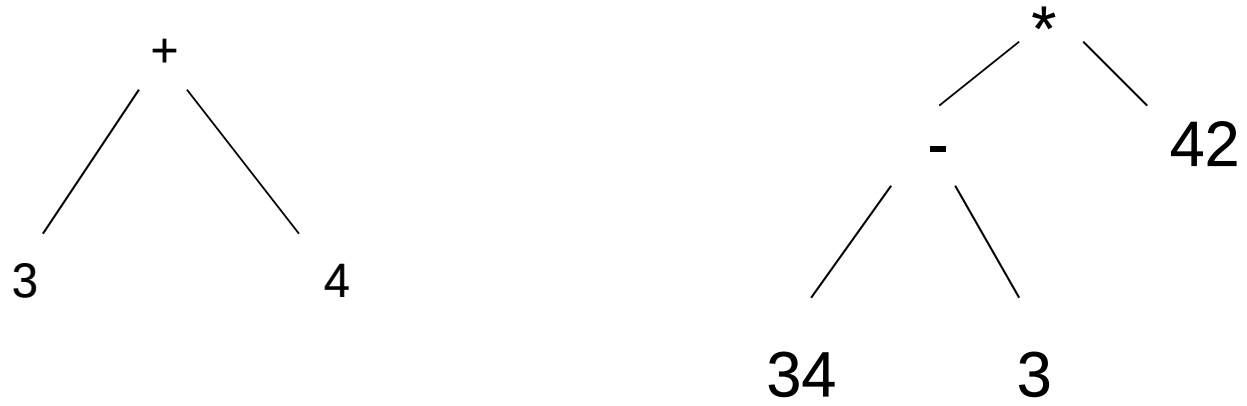


- Exp: **(num – num) * num**
- Tallene angir høyre-avledning
- Ser vi bort fra tallene, gir altså alle avlednings-rekkefølger det samme treet:



Abstrakt syntakstre

Vi tar bort de ”unødvendige” nodene i treet, de som stammet fra ”syntaktisk sukker” i språket. Vi sitter igjen med det som er den essensielle ”meningen” med setningen



Man ønsker ofte å skrive ned et tre på en sekvensiell fil:

Kan bruke prefiks eller postfiks form (eller innfiks eller ”omfiks”, eller ...)

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

  ** - 34 3 42* *”bruk strykejern BAKFRA!”*

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

- Postfiks form: *34 3 - 42 ** *”Bruk strykejern FORFRA”*

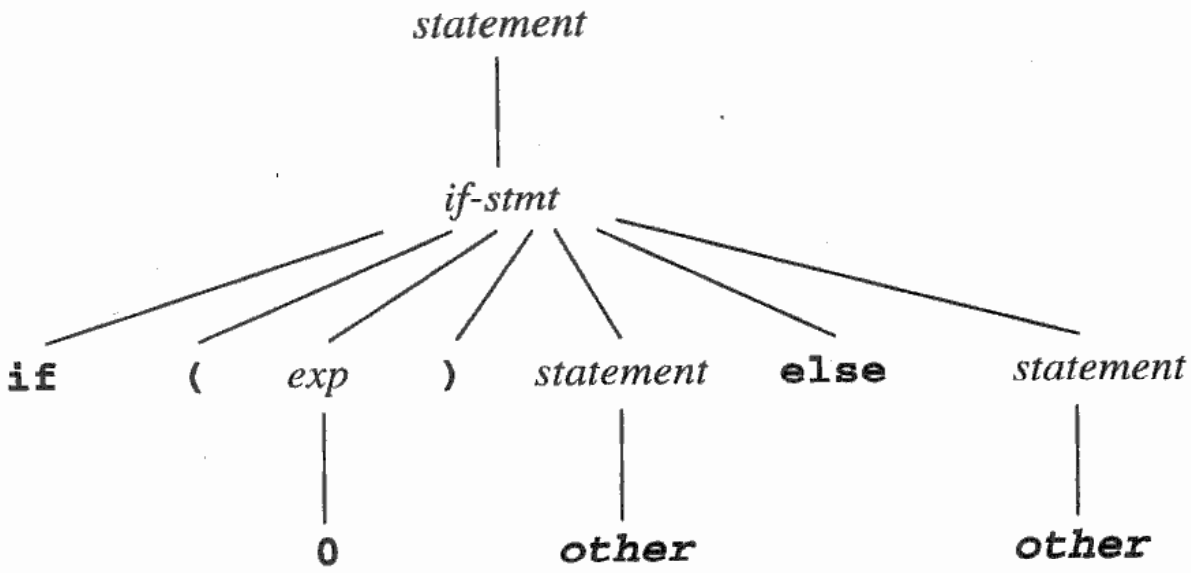
- Om det er *et kjent antall operander* for hver operator har disse entydige tolkning

Flere parserings-trær

G1:

statement → *if-stmt* | **other**
if-stmt → **if** **(** *exp* **)** *statement*
| **if** **(** *exp* **)** *statement* **else** *statement*
exp → **0** | **1**

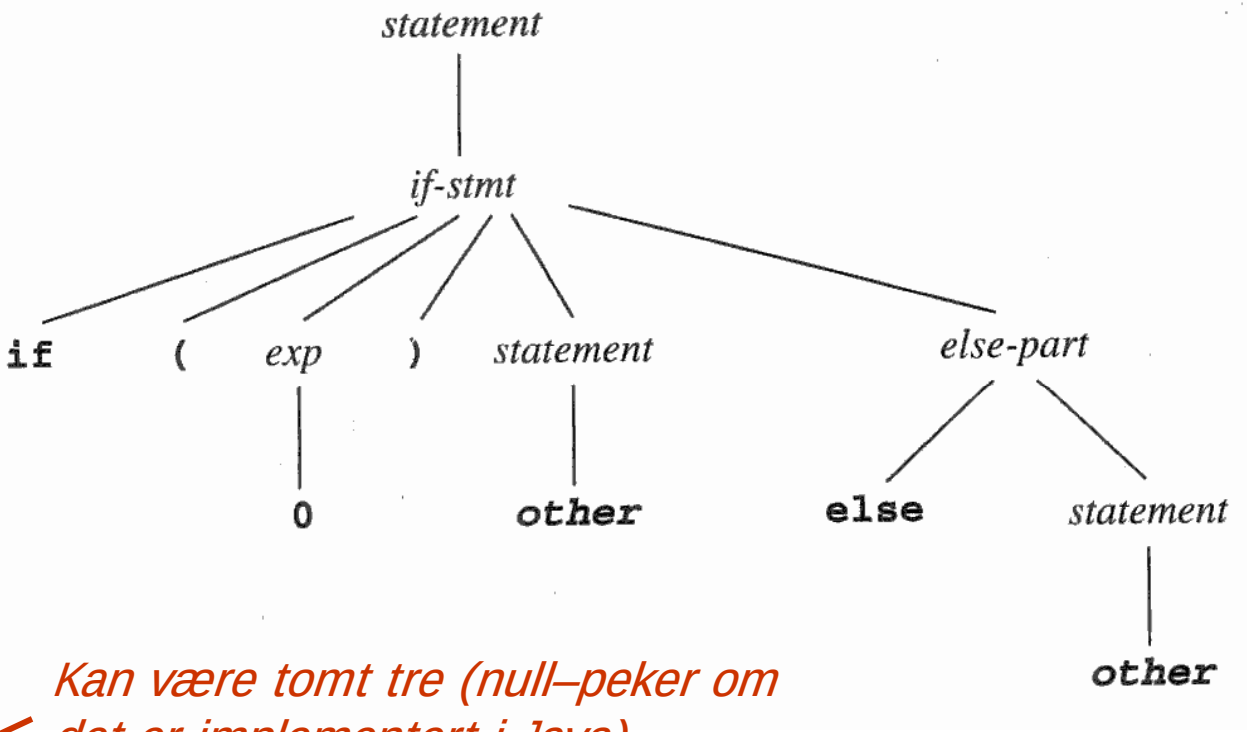
Setning: **if (0) other else other**



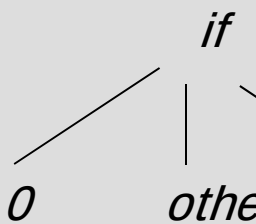
En annen grammatikk G2 for if-setninger

G2:

statement → *if-stmt* | **other**
if-stmt → **if** **(** *exp* **)** *statement else-part*
else-part → **else** *statement* | ε
exp → **0** | **1**



Felles abstrakt syntaks-tre for G1 og G2:



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

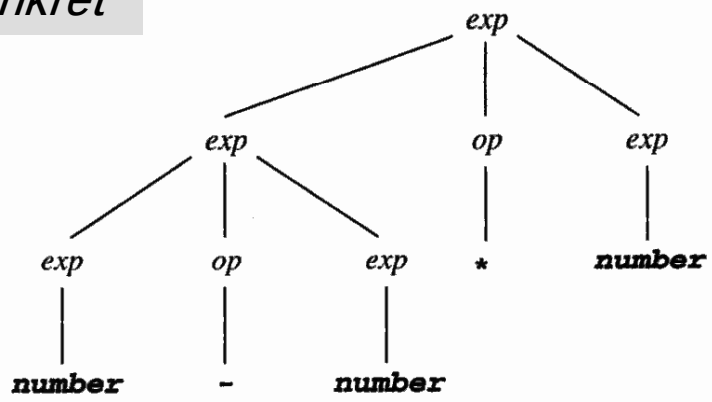
Flertydige grammatikker - analyse av setningen: n - n * n

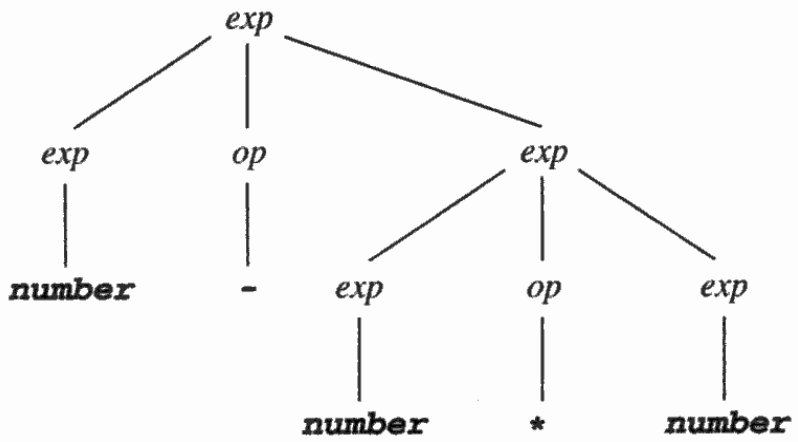
*G er flertydig hvis det finnes en setning i L(G) som kan gis **flere** parserings-trær*

De to trærne kan ofte angi helt forskjellige betydninger

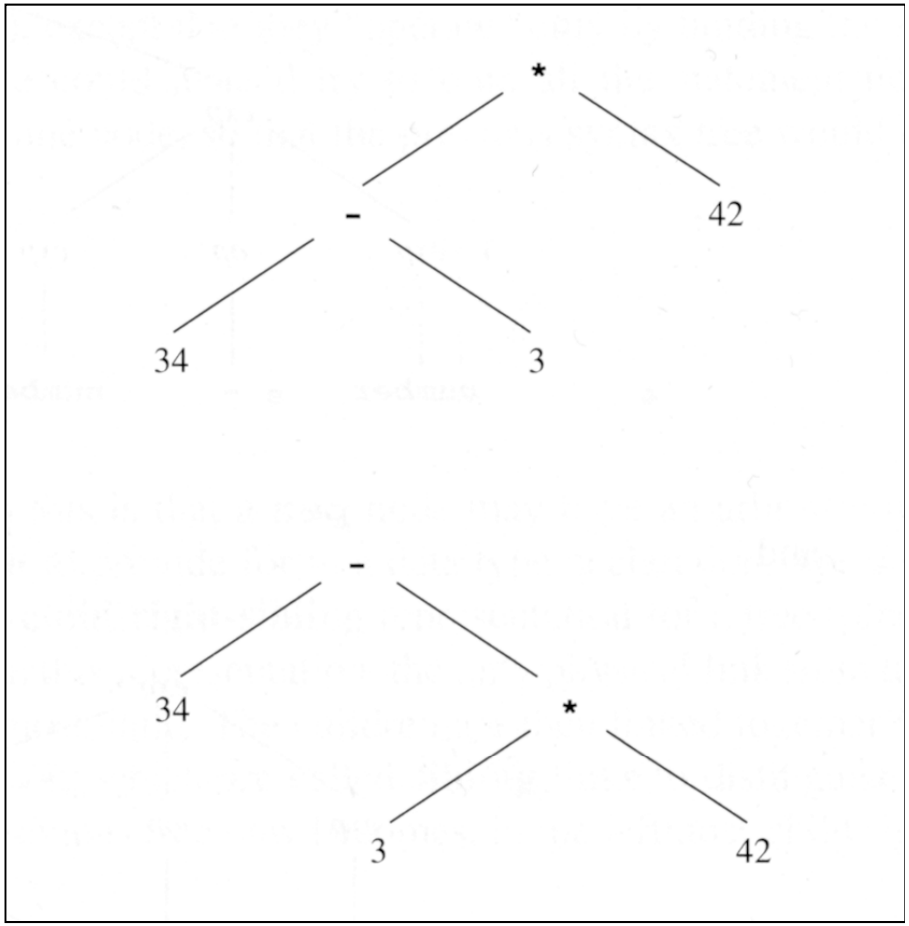
$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \texttt{number}$$
$$op \rightarrow + \mid - \mid *$$

Konkret





Abstrakt



Entydige grammatikker

$exp \rightarrow exp\ addop\ term \mid term$

$addop \rightarrow \texttt{+} \mid \texttt{-}$

$term \rightarrow term\ mulop\ factor \mid factor$

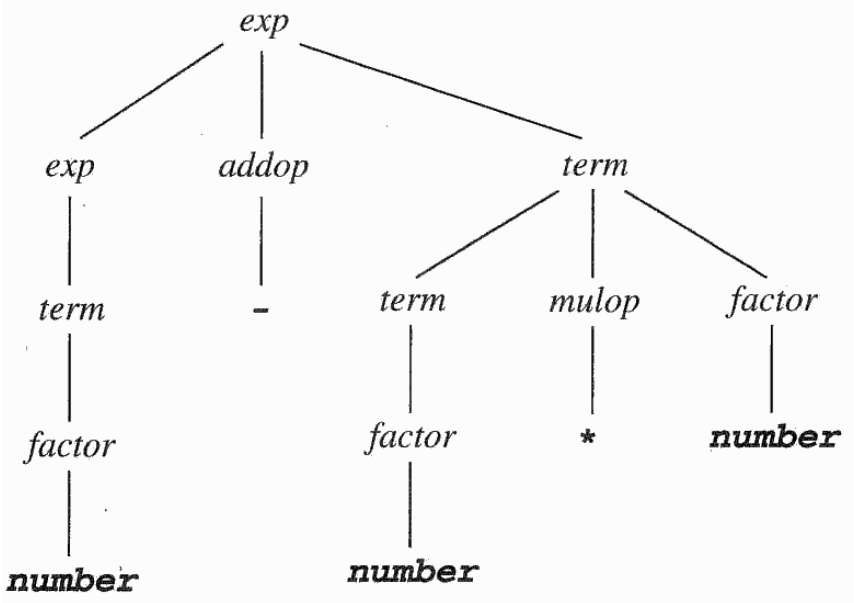
$mulop \rightarrow \texttt{*}$

$factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{number}$

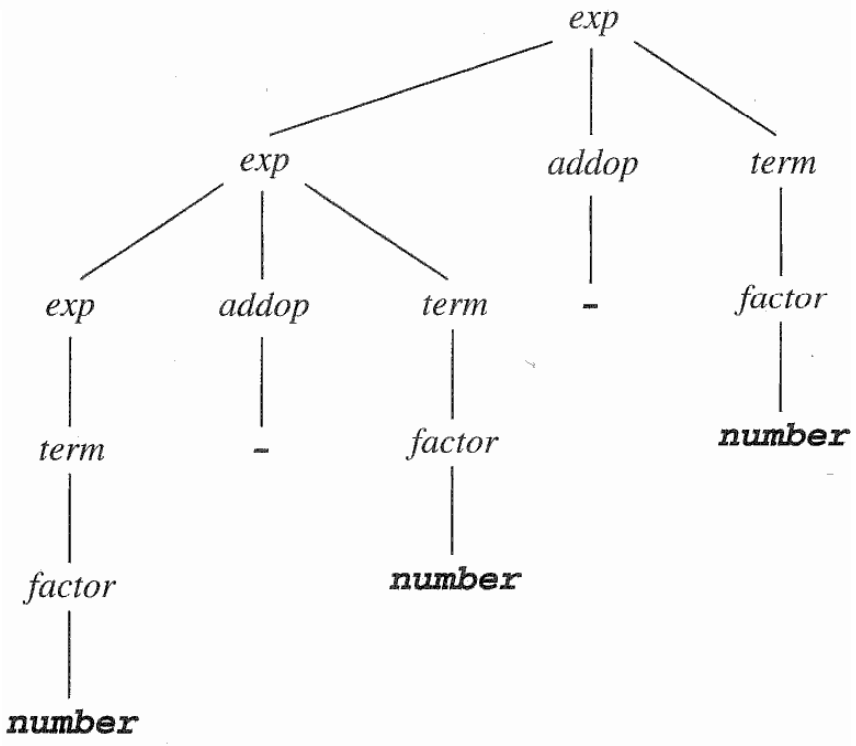
En ekstra ikke-terminal for hvert presedens-nivå

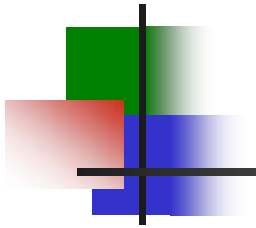
Rekkefølgen angir venstre/høyre assosiativitet

34-3*42



34-3-42





Presedens og assosiativitet uttrykt i en entydig grammatikk.

- Presedens for operatorer
  - Noen operasjoner skal gjøres før andre (* før +)
  - Ordnes ved en ekstra ikke-terminal for hvert presedensnivå (addop, multop, expop,..) – se forrige grammatikk
- Assosiativitet i grammatikken for operatorer:
  - Venstre assosiativitet: Operatorer med samme presedens utføres fra venstre mot høyre
  - Høyre assosiativitet: Operatorer med samme presedens utføres fra høyre mot venstre.
  - Ingen assosiativitet: Tillater ikke flere slike operasjoner etter hverandre (må bruke parenteser om det er mer enn to operander)
  - Venstre-assosiativitet ordnes ved at regler med slike operatorer gjøres "venstre-rekursive" :

    $exp \rightarrow exp\ addop\ term \mid term$
  - Høyre-assosiativitet Ordnes tilsvarende, men omvendt.
  - Ingen assosiativitet ordnes slik:

    $exp \rightarrow term\ addop\ term \mid term$

Ofte gjør man slik:

- Angir språket ved flertydige grammatikk som

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

- 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 ↑ 2 ↑ 3

Betyr: (3 + ((5 / 3) * 2)) + (4 ↑ (2 ↑ 3))

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

Presendens og assosiativitet i Java

Venstre assosiativ

Operator Precedence

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):

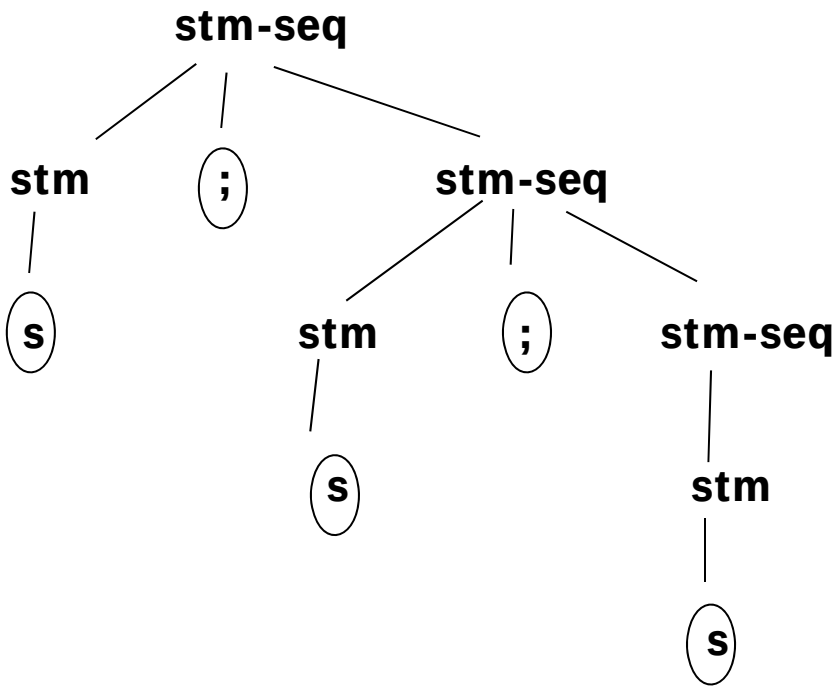
postfix ops	[] . (⟨exp⟩) ⟨exp⟩ ++ ⟨exp⟩ --
prefix ops	++⟨exp⟩ --⟨exp⟩ -⟨exp⟩ ~⟨exp⟩ !⟨exp⟩
creation/cast	**new** (⟨type⟩)⟨exp⟩
mult./div.	* / %
add./subt.	+ -
shift	<< >> >>>
comparison	< <= > >= **instanceof**
equality	== !=
bitwise-and	&
bitwise-xor	^
bitwise-or	|
and	&&
or	||
conditional	⟨bool_exp⟩?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

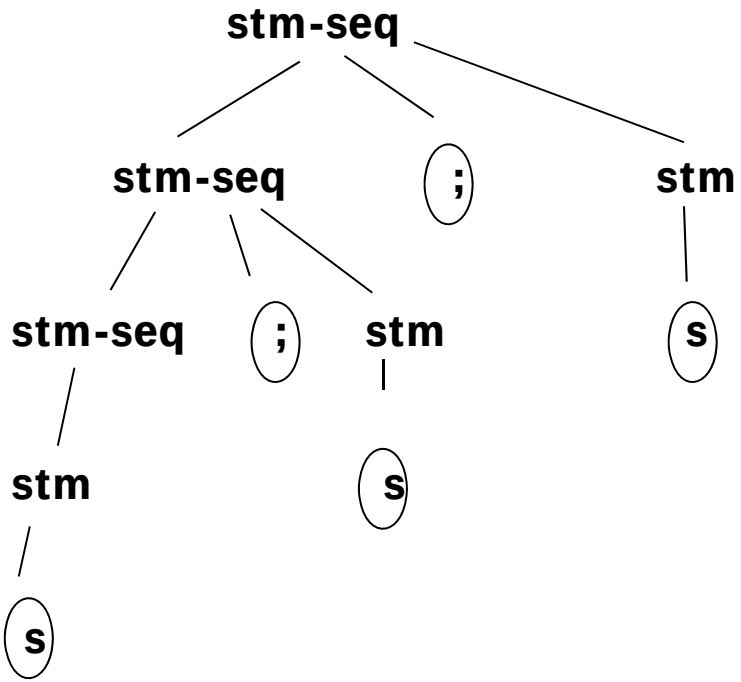
Ikke-essensiell flertydighet

stm-seq → stm-seq; stm | stm venstre-assos.

stm-seq → stm; stm-seq | stm høyre-assos.

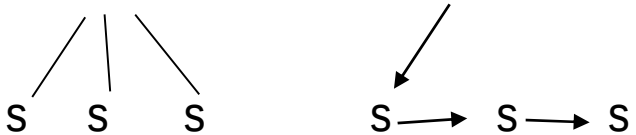
stm → s





Kan like gjerne representeres slik i praksis:

seq eller: seq



”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:

statement → *if-stmt* | **other**

if-stmt → **if** **(** *exp* **)** *statement*

| **if** **(** *exp* **)** *statement* **else** *statement*

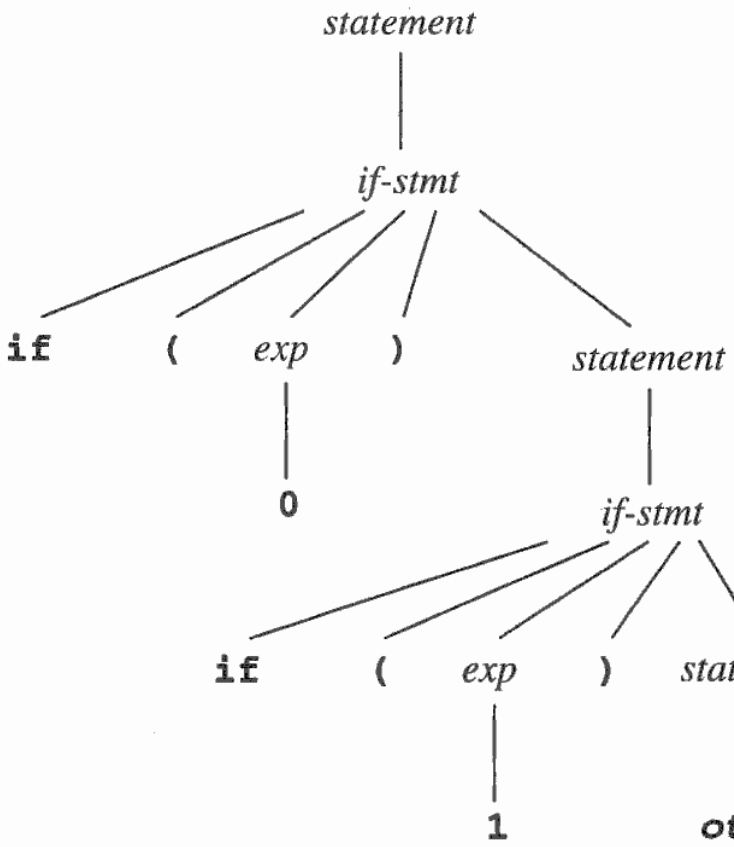
exp → **0** | **1**

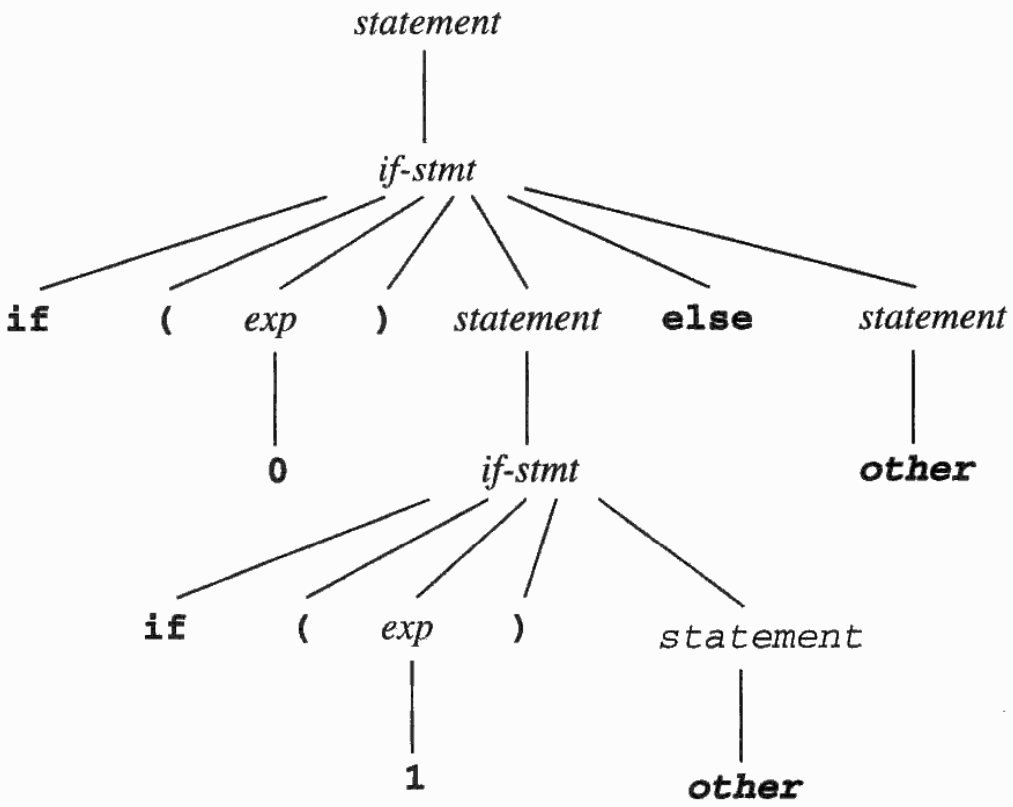
To trær for samme setning

statement → *if-stmt* | **other**
if-stmt → **if** **(** *exp* **)** *statement*
| **if** **(** *exp* **)** *statement* **else** *statement*
exp → **0** | **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 → *matched-stmt* | *unmatched-stmt*

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

unmatched-stmt → **if** **(** *exp* **)** *statement*
| **if** **(** *exp* **)** *matched-stmt* **else** *unmatched-stmt*

exp → **0** | **1**

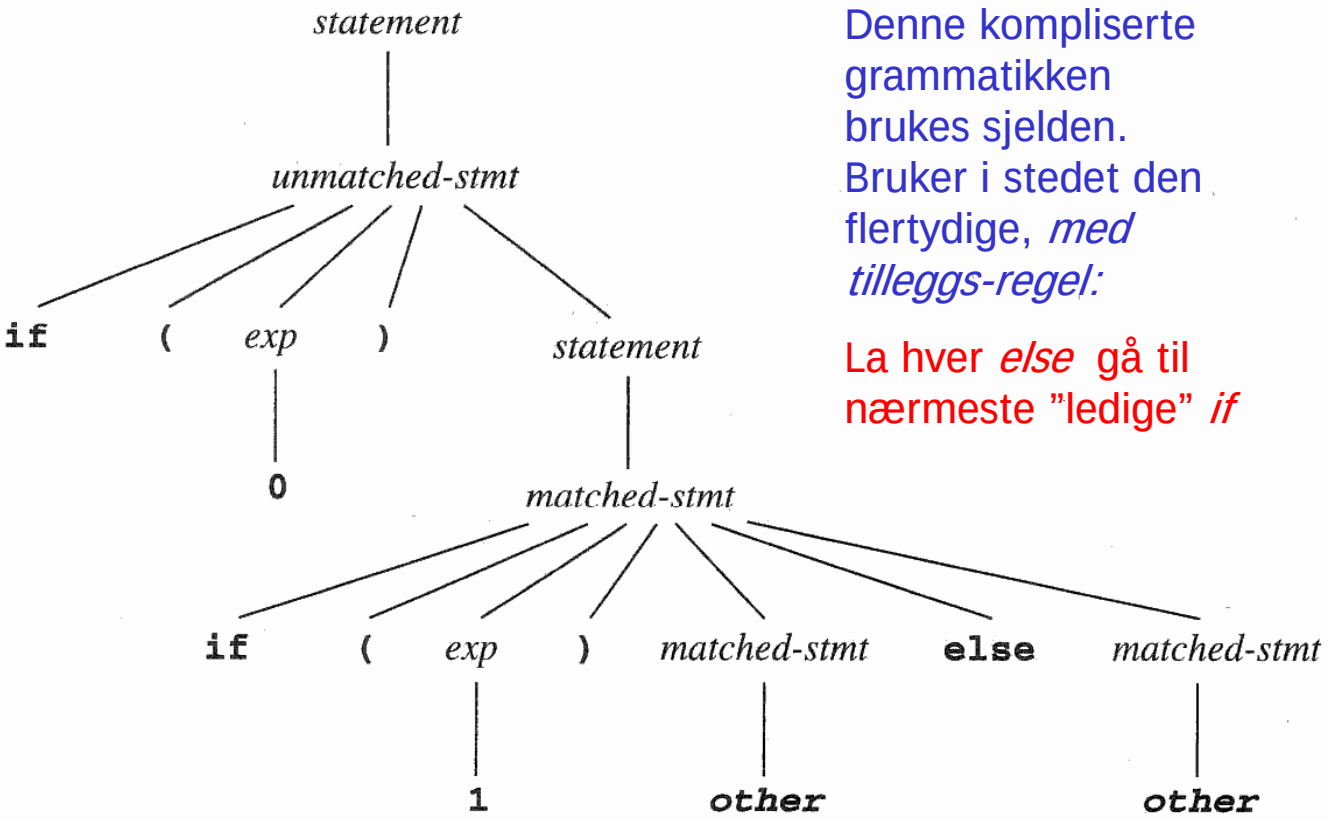
Idé:

Kan ikke ha en umatchet 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 opplagt 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: $\{\alpha\}$ α er en streng av terminaler og ikke-terminaler

$\alpha ?$ skrives: $[\alpha]$

Eksempel:

$$exp \rightarrow exp\ (\text{“+”} \mid \text{“−”} \mid \text{“*”})\ exp \mid \text{“(“}\ exp\ \text{“)”} \mid \textbf{number}$$

Meta-symbol ikke-meta

$A \rightarrow A\ \alpha \mid \beta$ kan skrives: $A \rightarrow \beta\ \{\alpha\}$

$A \rightarrow \alpha\ A \mid \beta$ kan skrives: $A \rightarrow \{\alpha\}\ \beta$

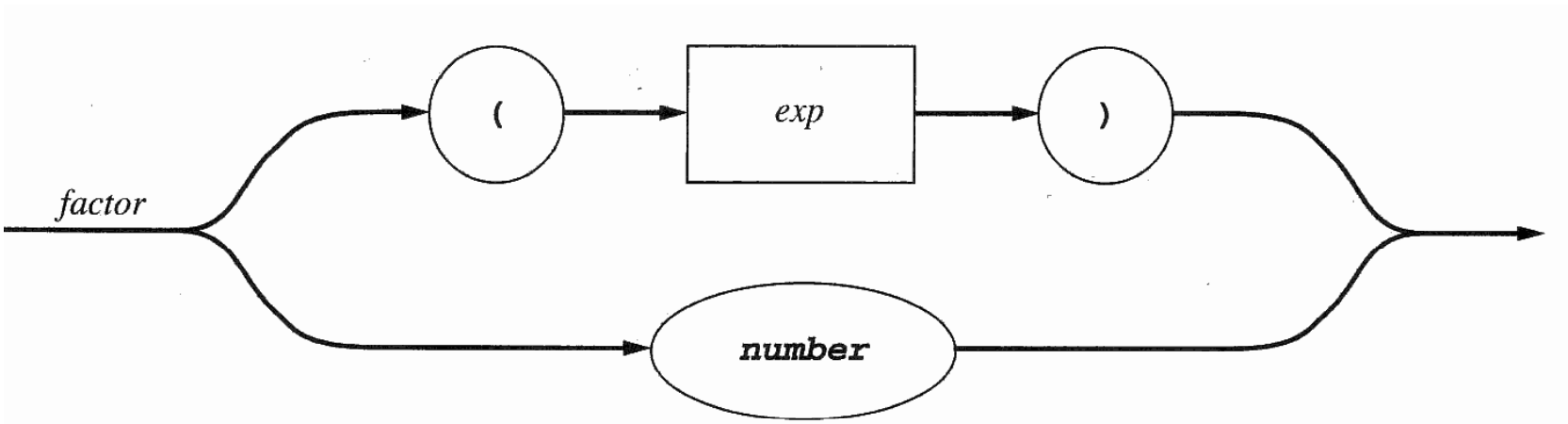
stm-seq → stm {; stm} eller stm-seq → {stm;} stm

if-setn → if (expr) stm [else stmt]

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

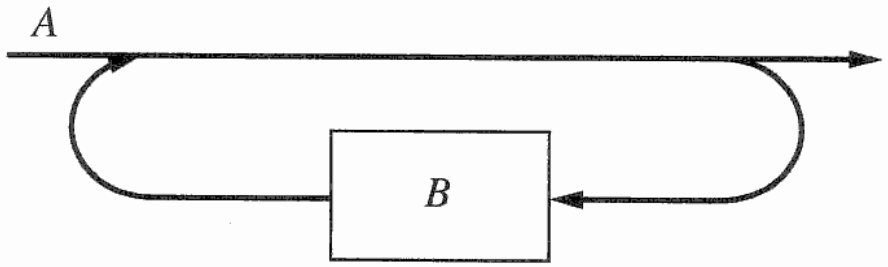
Syntaks-diagrammer

factor → (*exp*) | **number**

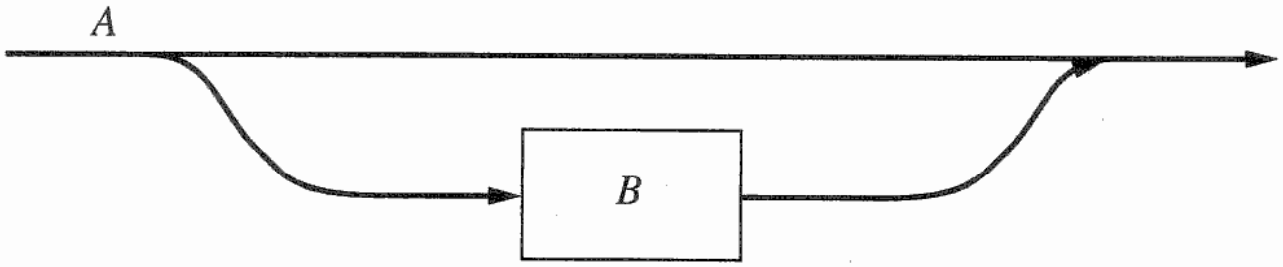


Merk: *"Entydige syntakstre"* og andre liknende begreper er ikke så naturlig å definere her

$$A \rightarrow \{\ B\ \}$$



$$A \rightarrow [\ B\ ]$$



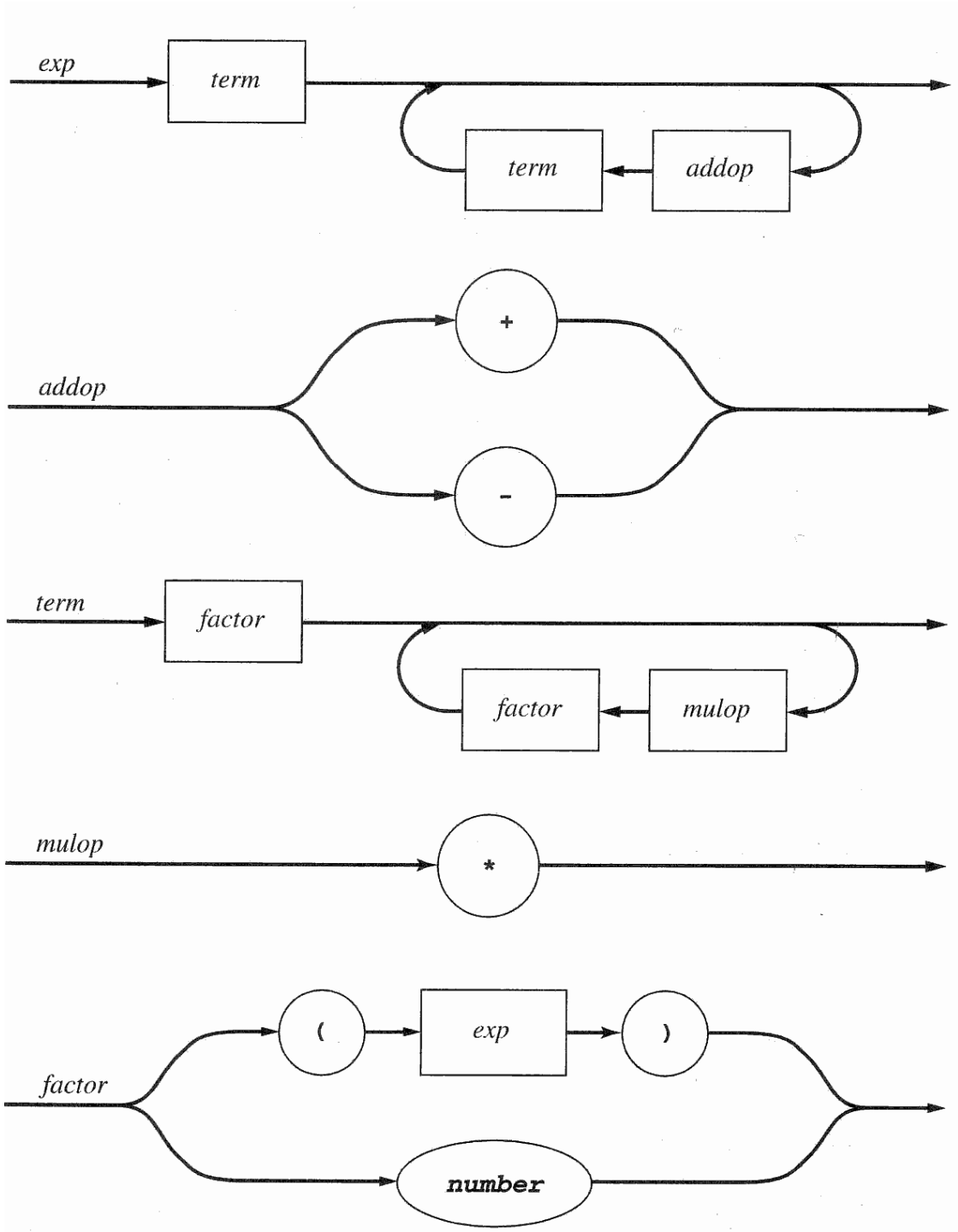
BNF-grammatikk:

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

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

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

Merk: Her må assososiativitet gis i tillegg

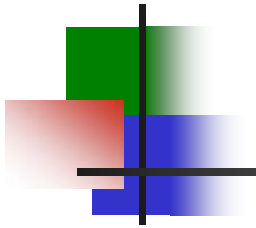


Chomsky-hierarkiet

a	er et vilkårlig terminalsymbol
β, α, γ	er vilkårlig streng av terminal- og ikke-terminalsymboler
A, B	er ikke-terminaler

- Type 0 – språk
  - Urestrikterte prod.: $\alpha \rightarrow \beta, \quad \alpha \neq \varepsilon$ (α 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 utrykk
    - 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?

- 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 sa at det skal være samme type på hver side av en tilording
- Vi bruker ikke grammatikker til å spesifiserer 'alt' ved språket, 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 deklarert
  - 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 syntakstræer til TINY

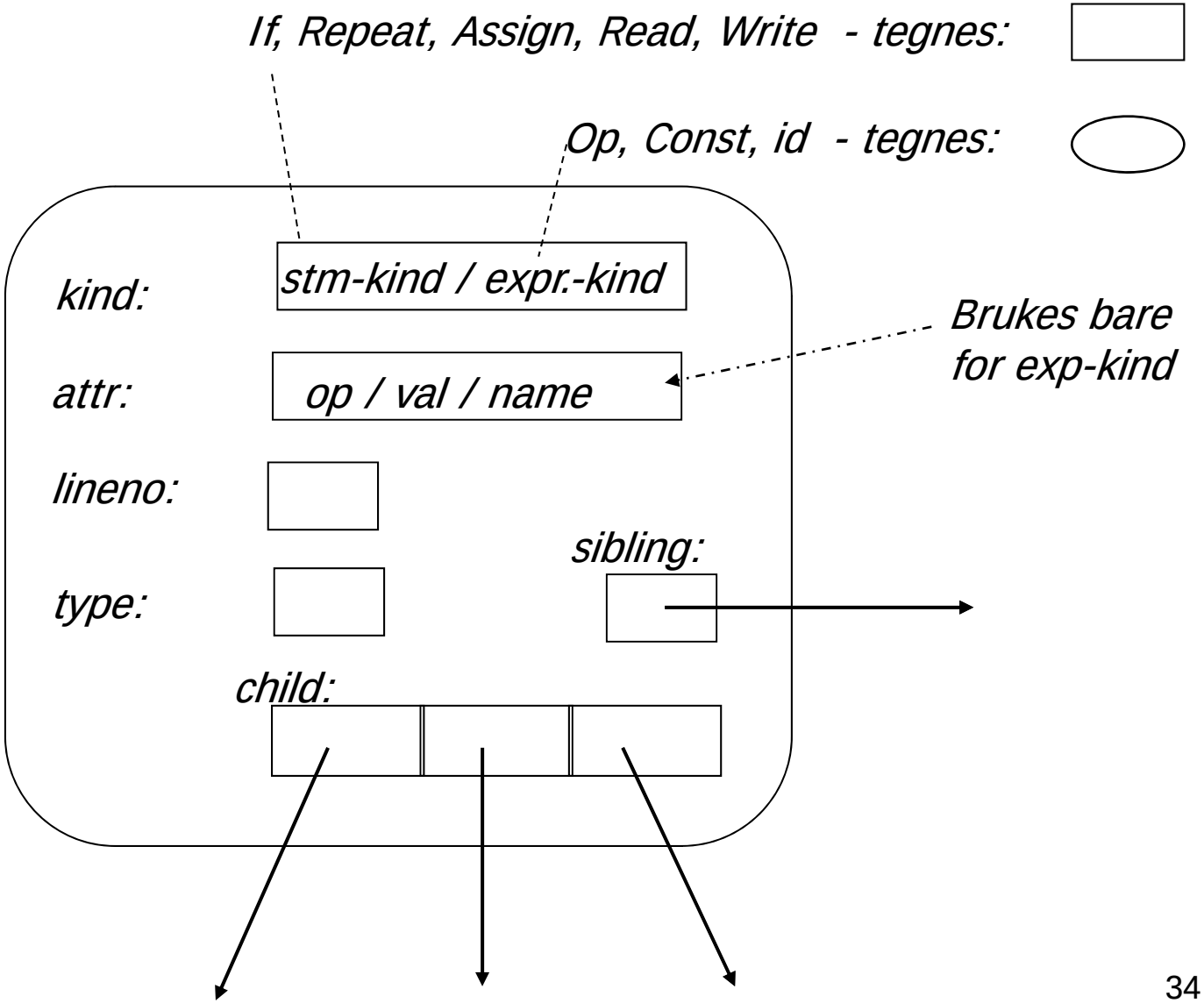
```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK}
             StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* 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;
```

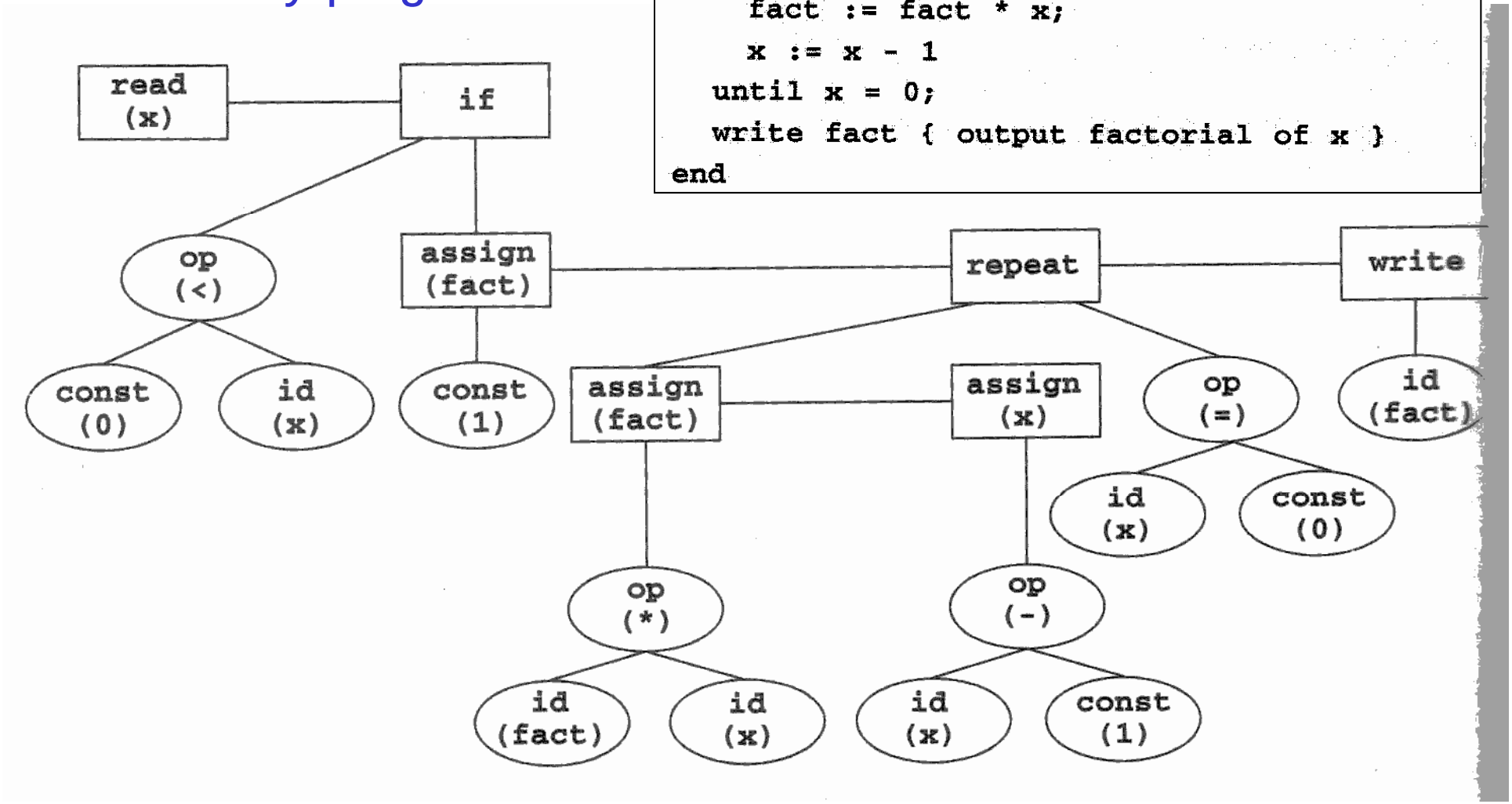
Nodestruktur i C for Tiny



Denne nodestrukturen passer enda bedre med et OO-språk med klasser og sublklasser **som implementasjons-språk.**
Da får vi et helt hierarki av klasser for nodene

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?*