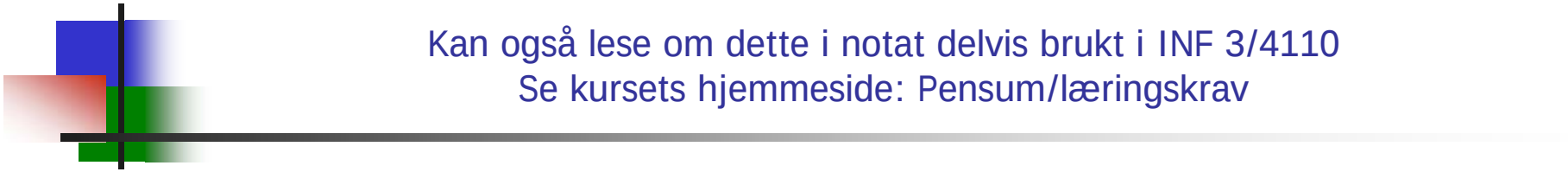
Kontekstfrie grammatikker og syntaksanalyse (parsering)

Kap. 3, 4 og 5 i Louden

Kan også lese om dette i notat delvis brukt i INF 3/4110
Se kursets hjemmeside: Pensum/læringskrav



1. februar 2007

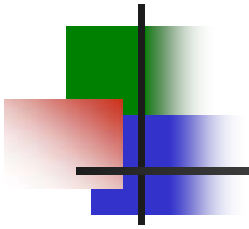
Stein Krogdahl
Ifi, UiO

Forelesninger fremover:

Tirsdag 6. februar: Vanlig forelesning

Torsdag 8. februar: Ikke forelesning

Tirsdag 13. februar: Vanlig forelesning



Om Stein Krogdahl

Utdannet:

- Ved UiO, Cand.Real. 1973
  - I ”Databehandling”, som da lå under Matematisk Institutt

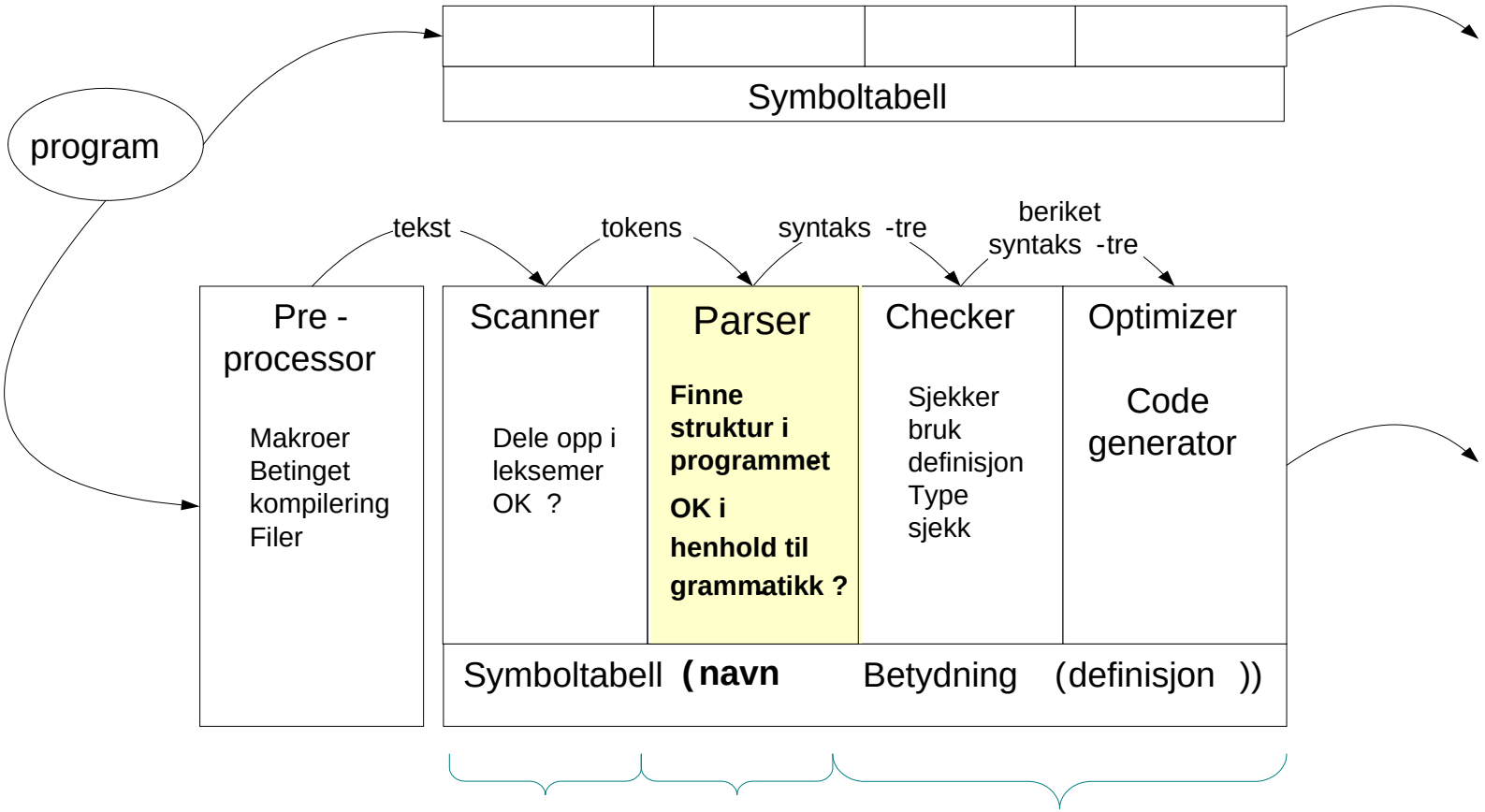
Har vært ved:

- Universitetet i Tromsø (1973 – 78)
- Norsk Regnesentral (1978 – 82)
  - Laget Simula-kompilator med Birger M-P
- Universitetet i Oslo, Ifi (1982 – nå)

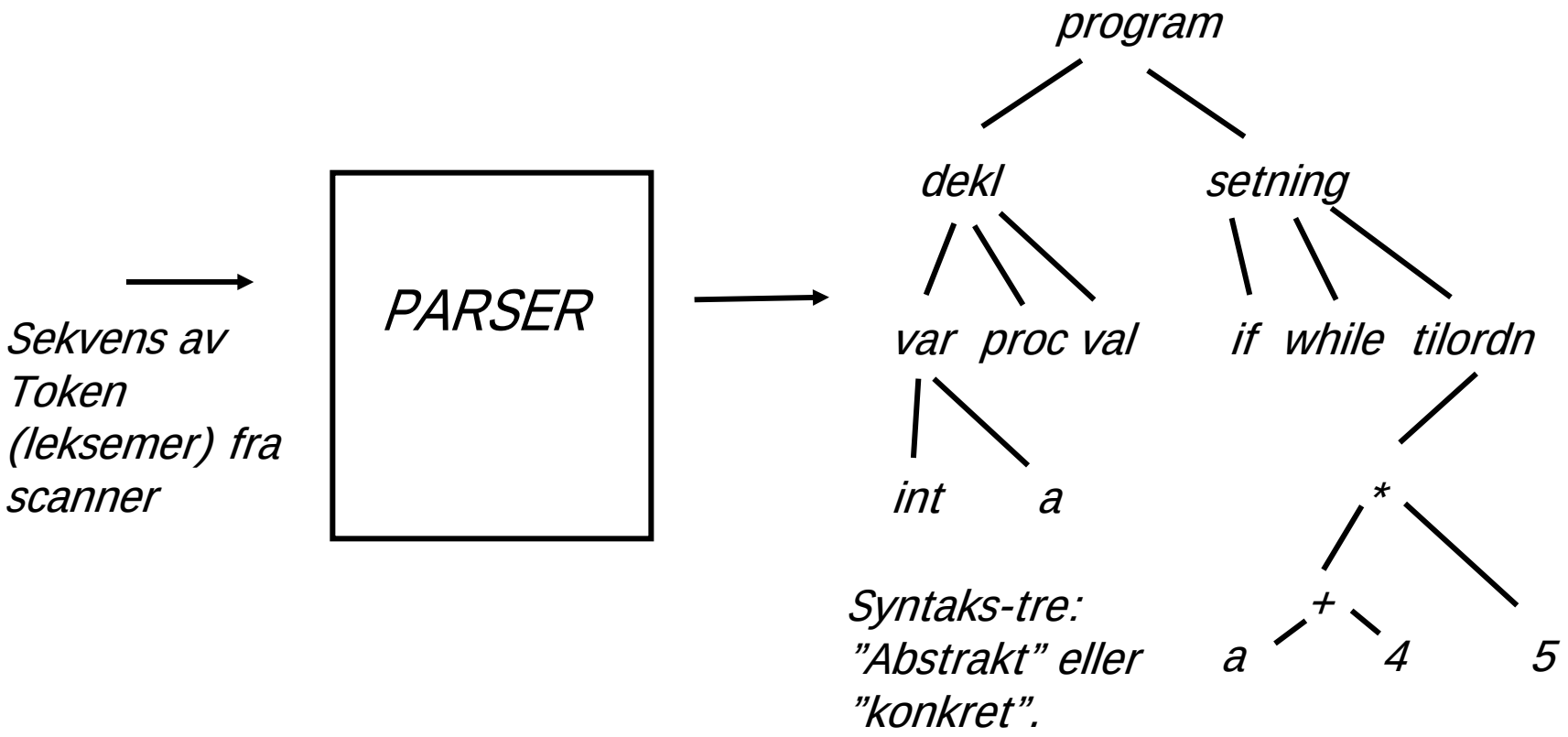
Har drevet med

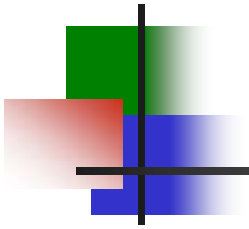
- Kombinatoriske algoritmer
- Verifikasjon av programmer (av mer intuitiv art)
- Programmeringsspråk: Design og implementasjon
- Nå: Mest SWAT-prosjektet.

Hvor er vi nå - kap. 3, 4 og 5:



Forenklet skisse av hva en parser gjør



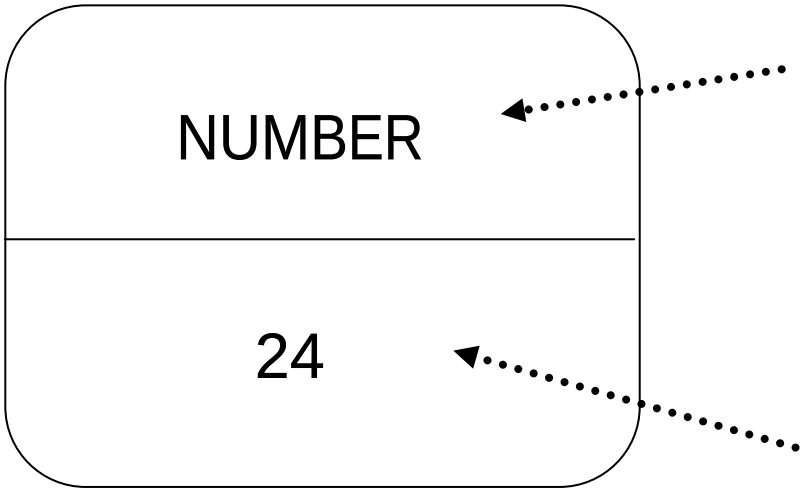


Oversikt – kap 3 (grunnleggende om grammatikker)

- Hva er en grammatikk
- Vi kommer til å lære flere (mange) typer grammatikk
  - hvorfor ?
- Kontekstfrie grammatikker
- Parserings-trær og abstrakte syntaks-trær
- Tvetydige grammatikker
- Utvidet notasjon: EBNF og syntaksdiagrammer
- Eksempel
  - Tiny

Hva får vi fra scanneren

En sekvens av slike:



Token
(type av symbol, det som analyseres i parseren)

Leksem

(det vi skrev i programmet)

- Av og til vil vi gi eksempler med token og av og til med leksem.

- Ofte kalles også det hele for et token,

- I forbindelse med parsering kalles det også et "terminalsymbol" (eller bare et symbol)

Hva er en grammatikk

- En grammatikk bruker et alfabet
  - Ofte det vi kan taste inn fra tastaturet
  - Her: Mengden av token-typer som kommer fra scanneren
- En grammatikk består av noen **symboler** = sammensetninger av tegn fra alfabetet:
  - **Terminal-symboler** - slike vi bruker når vi programmerer
    Dvs. for parseren er det slike tokens/leksemer vi får fra skanneren:
    - int, TALL, if , VARIABEL, ...
  - **Ikke-terminalsymboler** - begreper vi bruker i grammatikken):
    - WHILE-setning, TILORDNING, KLASSEDEKL,UTTRYKK...
    - Startsymbolet : Lovlige setninger er utledet fra dette.
  - **Meta-symboler**- slike hjelpesymboler/tegn vi bruker for å sette opp reglene:
- En grammatikk spesifiserer via regler lovlige sammensettinger av terminal- og ikke-terminalsymboler:

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

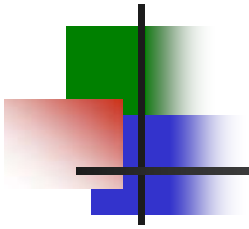
Grammatikk - eksempel

- Grammatikken:

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

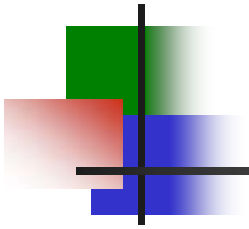
- Metasymboler: ::=, <,>,|
- Ikke-terminaler: exp, op
- Terminaler : NUMBER, (,), *, +, -

::= ' leses som:'	kan bestå av
|	'betyr'	eller



Rollen til en grammatikk, semantikk og syntaks

- En grammatikk definerer *visse sider* av et språk – **syntaksen** (form-reglene)
  - Gjelder naturlige språk (tysk,.., norsk)
  - Gjelder alle kunstige språk som programmeringsspråk
  - Ikke alle de setningene vi kan lage i grammatikken, gir mening
- **Semantikken** (meningen) med de ulike delene spesifiseres separat – se kap.6.
- En grammatikk definerer regler for rekkefølgen av terminalsymboler som er lovlige setninger (programmer) i språket
  - Med en grammatikk kan man lage (avlede) mange, oftest uendelig mange, programmer
    - Selv om et program er riktige etter grammatikken, vil det ofte ikke tilfredsstille de andre (”statisk semantiske”) regler for et riktig program.
    - Eksempel: int i = true;



Hvorfor ikke bare én (stor) grammatikk?

- Kunne vi ikke laget en (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 typen grammatikker vi bruker
    - F.eks. at alle variable er deklarert
  - mye greiere å ta:
    - enkle ting i skanneren,
    - setningsformen i parseren
    - mer kompliserte krav i semantikk-sjekkeren
      - 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

Kontekstfrie grammatikker, BNF notasjon med variasjoner

- Bokas vanlige notasjon

  exp → exp op exp | (exp) | number

  op → + | - | *

- En tradisjonell måte (Algol 60 rapporten)

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

- Litt utvidet BNF

exp → *exp* (“**+**” | “**–**” | “*****”) *exp* | “**(**“ *exp* ”**)**” | ***number***

Flere måter å skrive den samme grammatikken

- Regnes som den mest basale:

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

- Kortest mulig

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

Avledning (venstreavledning) av: **(number – number) * number**

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

	Startsymbol	Produksjon (regel) 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 = avleder med terminaler fra venstre
Ferdig når det bare er terminalsymboler

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

grammatikk *streng med bare terminal-symboler*

Avledning (høyreavledning)

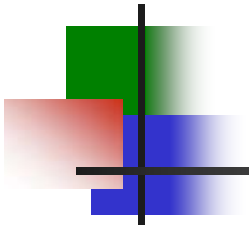
av: (number – number) * number

$exp \rightarrow exp\ op\ exp$
$exp \rightarrow$ **(** 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 = avleder med terminaler fra høyre
Ferdig når det bare er terminalsymboler

Det finnes også mange andre rekkefølger å avlede en setning fra en grammatikk



Opplagte krav til en 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:

  $A \rightarrow Bx$

  $B \rightarrow Ay$

  $C \rightarrow z$

  - Kan aldri avlede fra A til bare terminalsymboler
  - C kan ikke inngå i noen streng avledet fra A

  Altså en håpløs grammatikk

Parserings-tre (konkret syntaks-tre)

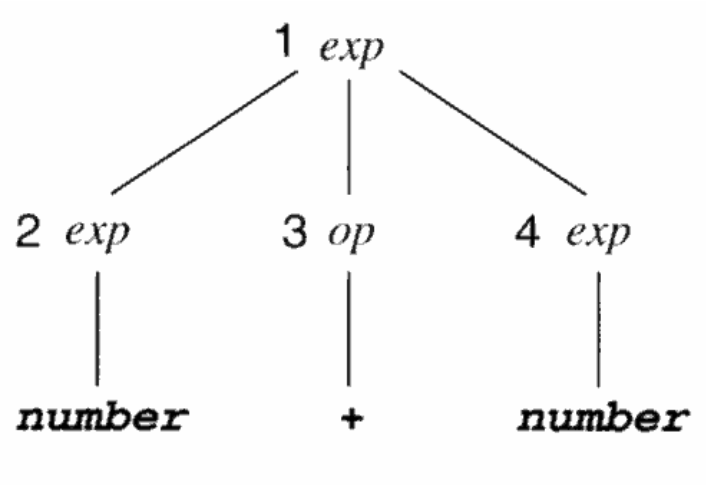
(1) $exp \Rightarrow exp\ op\ exp$

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

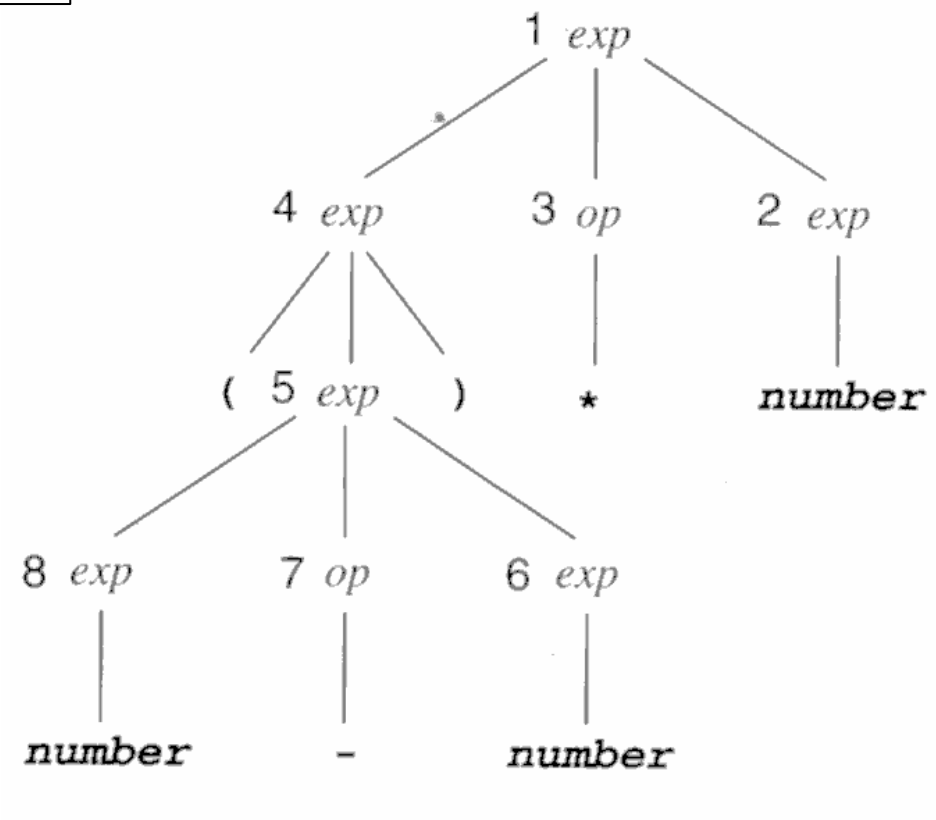
(3) $\Rightarrow$ **number** + exp

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

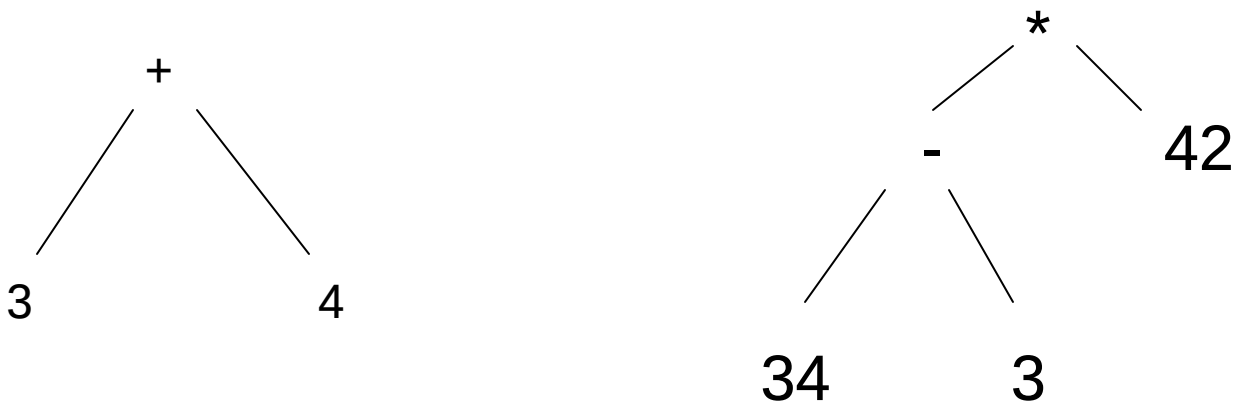
- Representasjon som er uavhengig av avlednings-rekkefølgen



- Tallene angir høyre/avledning
- Ser vi bort fra tallene, gir alle avlednings/rekkefølger det samme treet



(Abstrakt) syntakstre – forlengs polsk notasjon
- “fra tre til fil-format” – det vi egentlig trenger videre



*Prefix-form av treet for (34-3)*42: (”veltet til venstre” – prefiks traversering – først noden, så venstre sub-tre, så høyre sub-tre):*

OpExp(Times, OpExp(Minus, Const(34), Const(3)), Const(42))

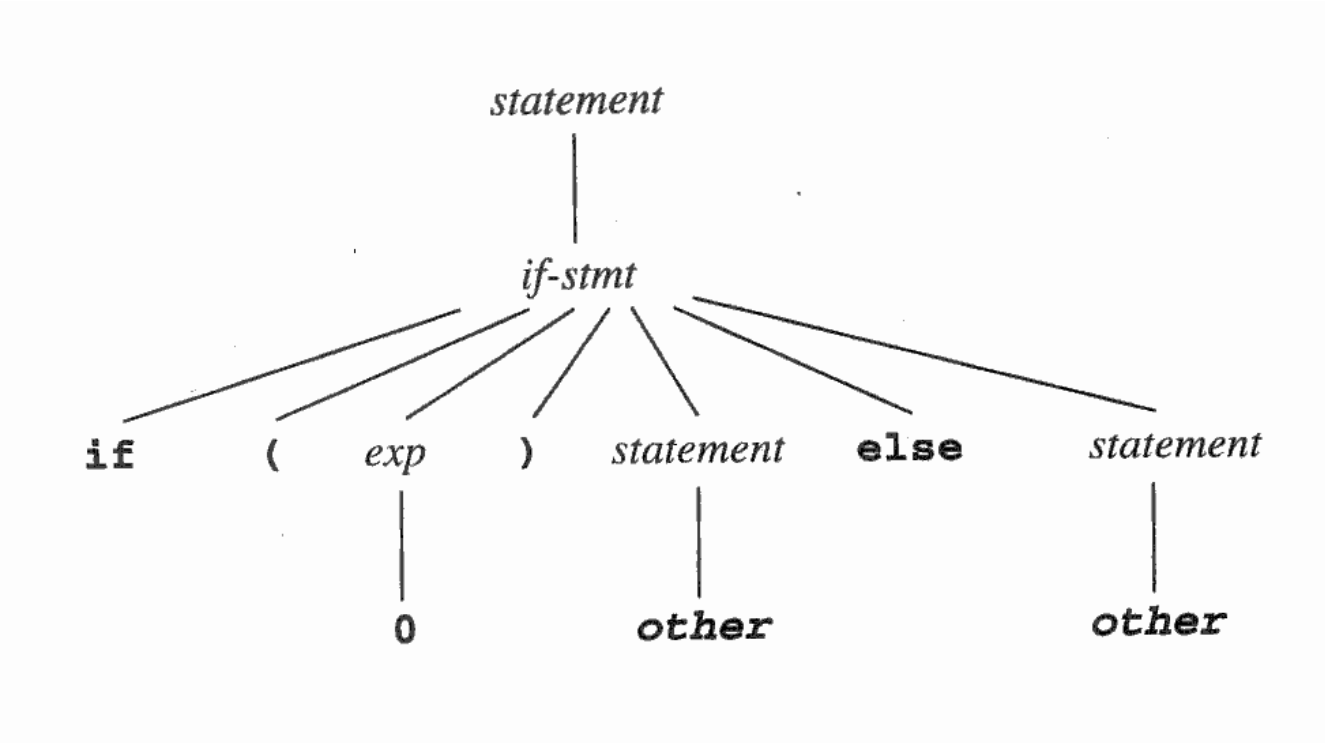
Flere parserings-trær

G1:

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

Setning:

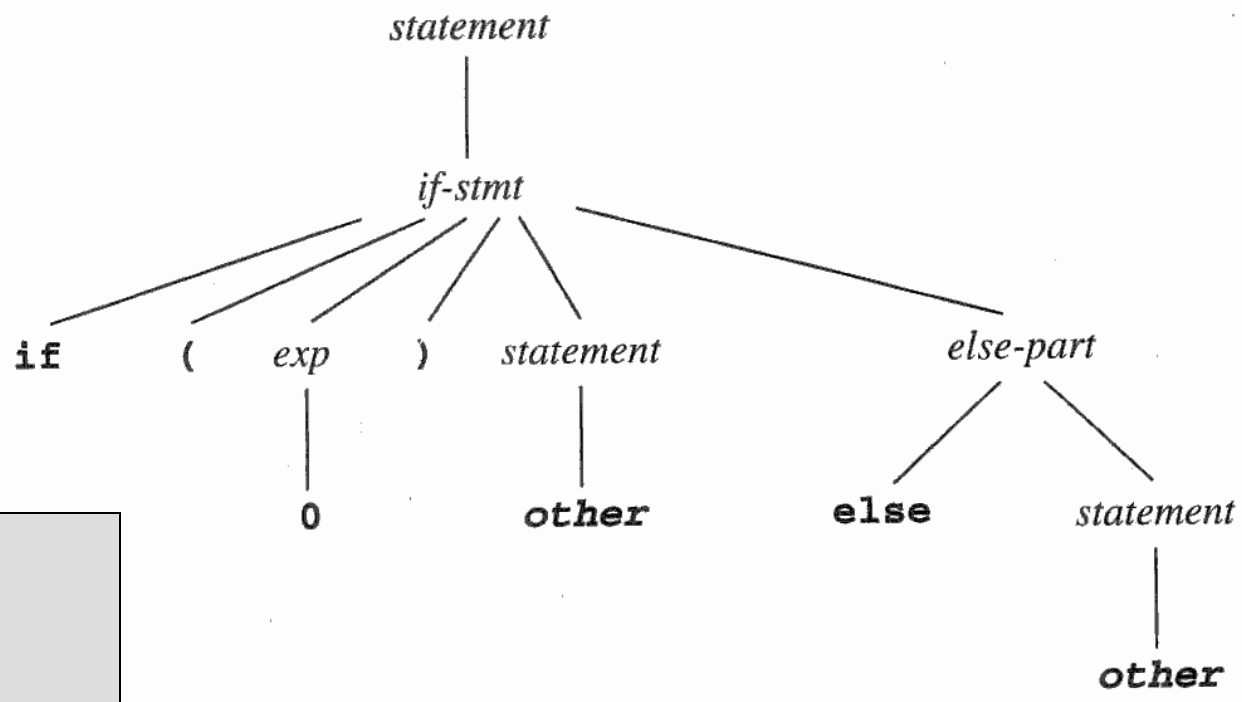
if (0) other else other



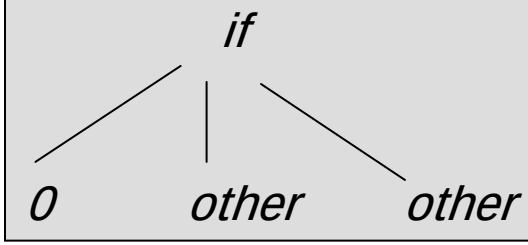
En annen grammatikk G2 for if-setninger

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:



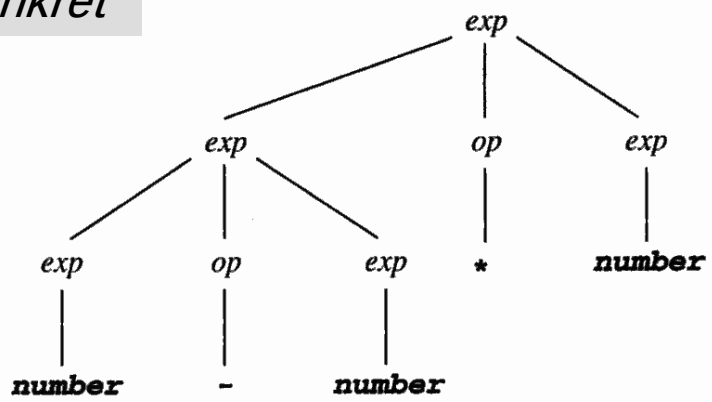
Tvetydige grammatikker - analyse av setningen: n - n * n

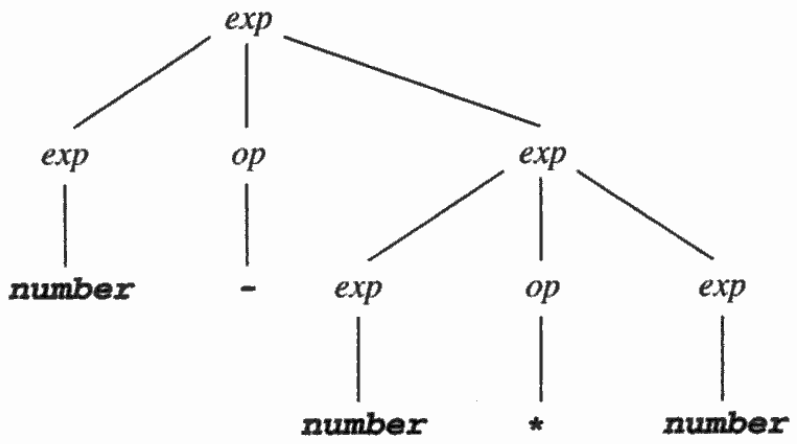
*G er flertydig hvis det finnes en setning i L(G) som kan gis **flere** parserings-trær*

Dette eksempelet er essensiell tvetydighet, angir ulike beregninger.

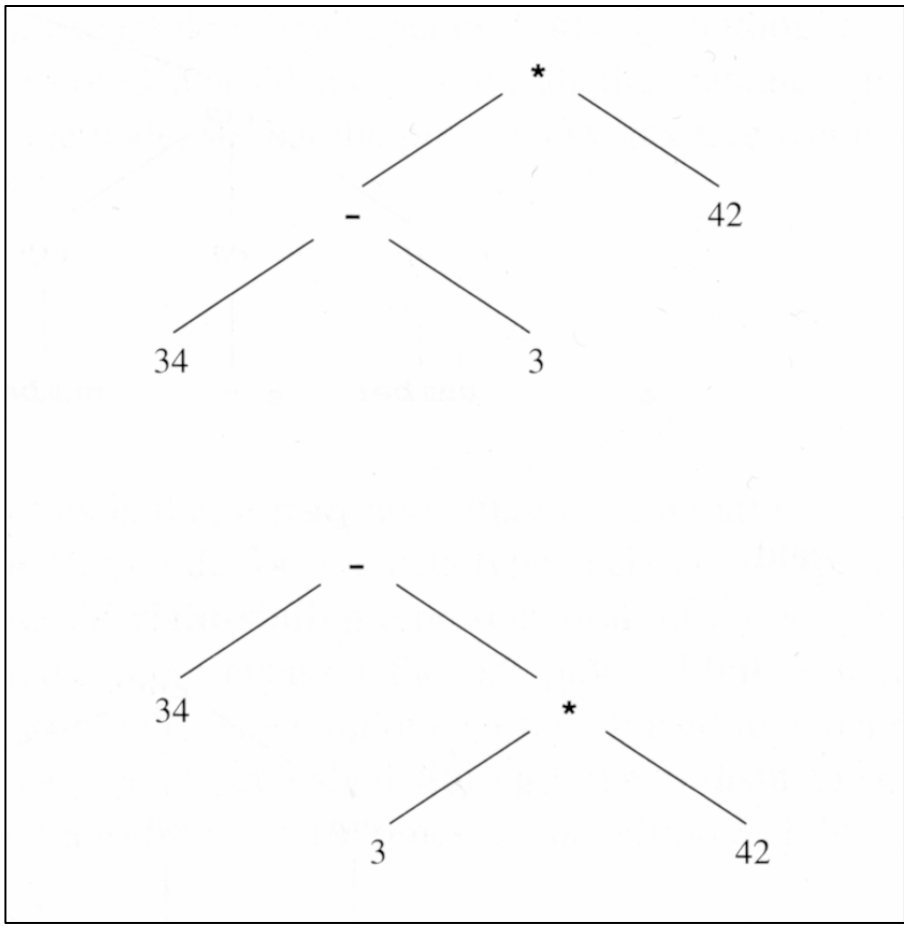
$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \textbf{number}$$
$$op \rightarrow + \mid - \mid *$$

Konkret





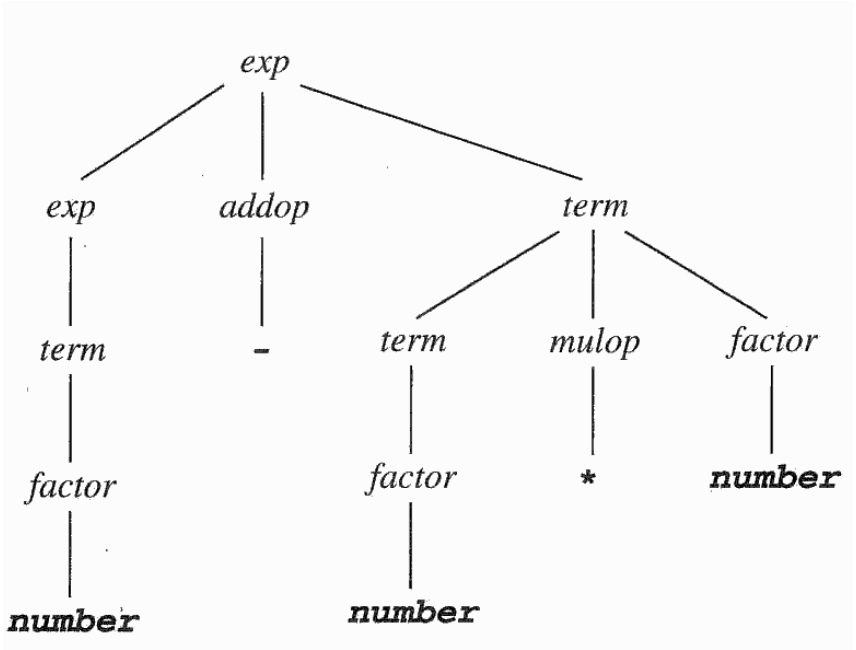
Abstrakt



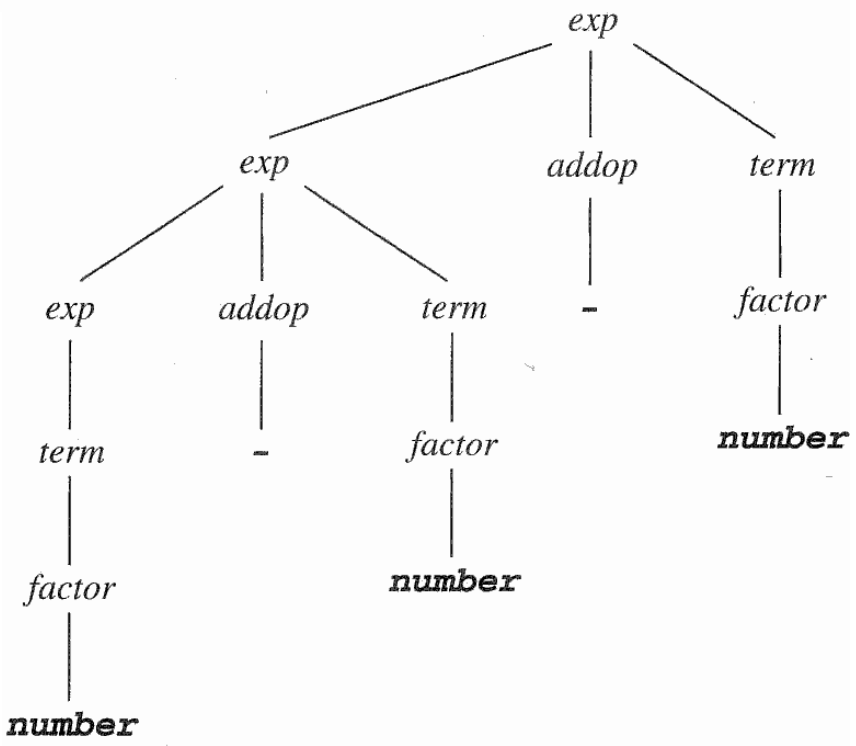
Entydige grammatikker for samme språket, to trær for to *ulike* uttrykk.

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

34-3*42



34-3-42



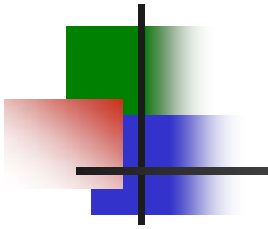
To eksempler på viktigheten av assosiativitet (og presedens)

a) Melding til lederen for en eksekusjons-peletong:

- Stopp ikke skyt ham !
  - Stopp ikke - skyt ham
  - Stopp - ikke skyt ham

b) uttrykket : 3 - 4 - 5

- 3 - (4-5) = 3-(-1) = 4 høyre
- (3-4) -5 = - 6 venstre



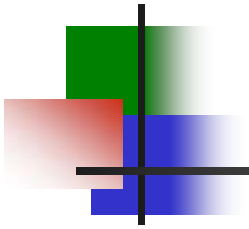
Presedens og assosiativitet uttrykt i grammatikken

- Presedens i grammatikken for operatorer
  - Noen operasjoner gjøres før andre (* før +)
  - Ordnes med kaskading av definisjoner av operatorgrupper (addop, multop, expop,..) – se forrige grammatikk
- Assosiativitet i grammatikken for operatorer:
  - Venstre (assositivitet) : Operatorer med samme presedens utføres fra venstre mot høyre
  - Høyre (assositivitet) : Operatorer med samme presedens utføres fra høyre mot venstre.
  - Ingen (assositivitet) : Tillater ingen rekkefølge av slike operasjoner. i samme .
  - Ordnes med at uttrykk med slike operatorer bare kan utvides på den ene siden :

    exp $\rightarrow$ *exp addop term* | *term*

    (denne utvider på venstre side og blir venstre assisiativ)

    Ingen assosiativitet ordnes med at vi krever parenteseser med bare en operator i en parentes.



Vanlig å:

- Angi språket ved flertydige grammatikk som

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

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

  +, lav, venstre ass.

  *, høy , venstre-ass

  ↑ , høyerst, høyre ass.

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

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

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):

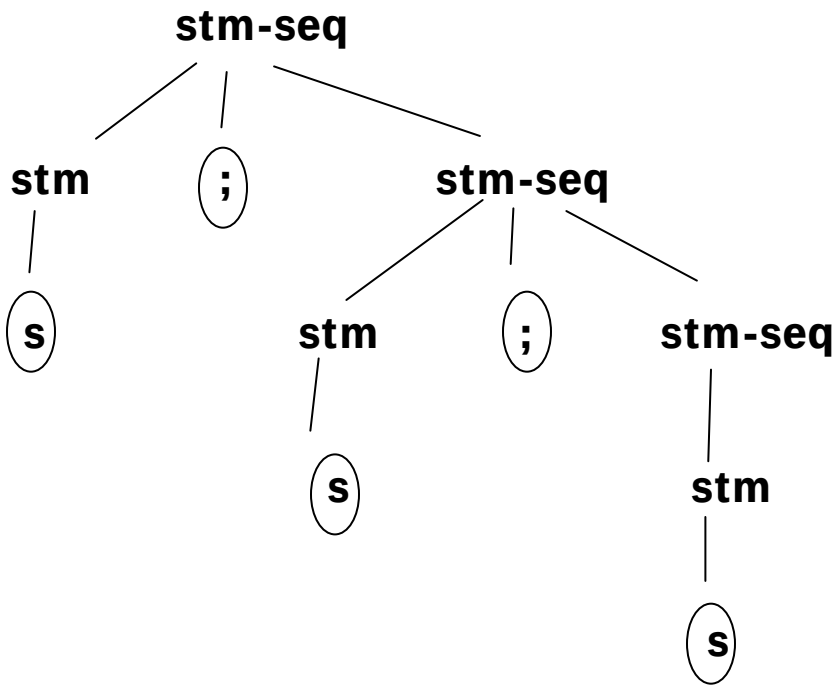
postfix ops	[] . (⟨exp⟩) ⟨exp⟩ ++ ⟨exp⟩ --
prefix ops	++⟨exp⟩ --⟨exp⟩ -⟨exp⟩ ~⟨exp⟩ !⟨exp⟩
creation/cast	**new** (⟨type⟩)⟨exp⟩
mult./div.	* / %
add./subt.	+ -
shift	<< >> >>>
comparison	< <= > >= **instanceof**
equality	== !=
bitwise-and	&
bitwise-xor	^
bitwise-or	|
and	&&
or	||
conditional	⟨bool_exp⟩?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

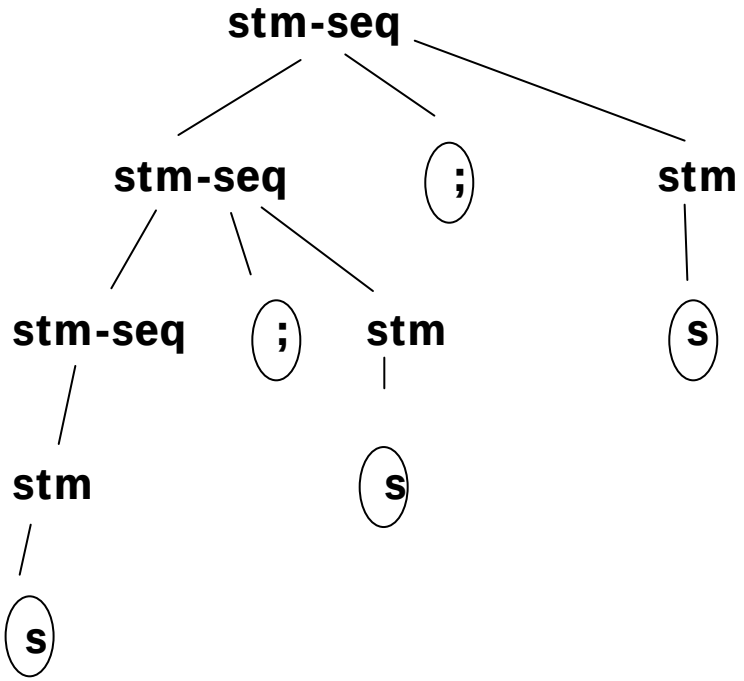
Ikke-essensiell flertydighet

stm-seq → stm-seq; stm | stm *venstre-ass*

stm-seq → stm; stm-seq | stm høyre-ass

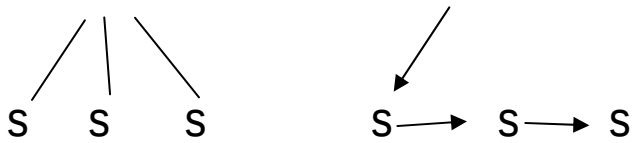
stm → s





Kan like gjerne representeres som:

seq eller: seq



”Dangelig else” - problemet

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

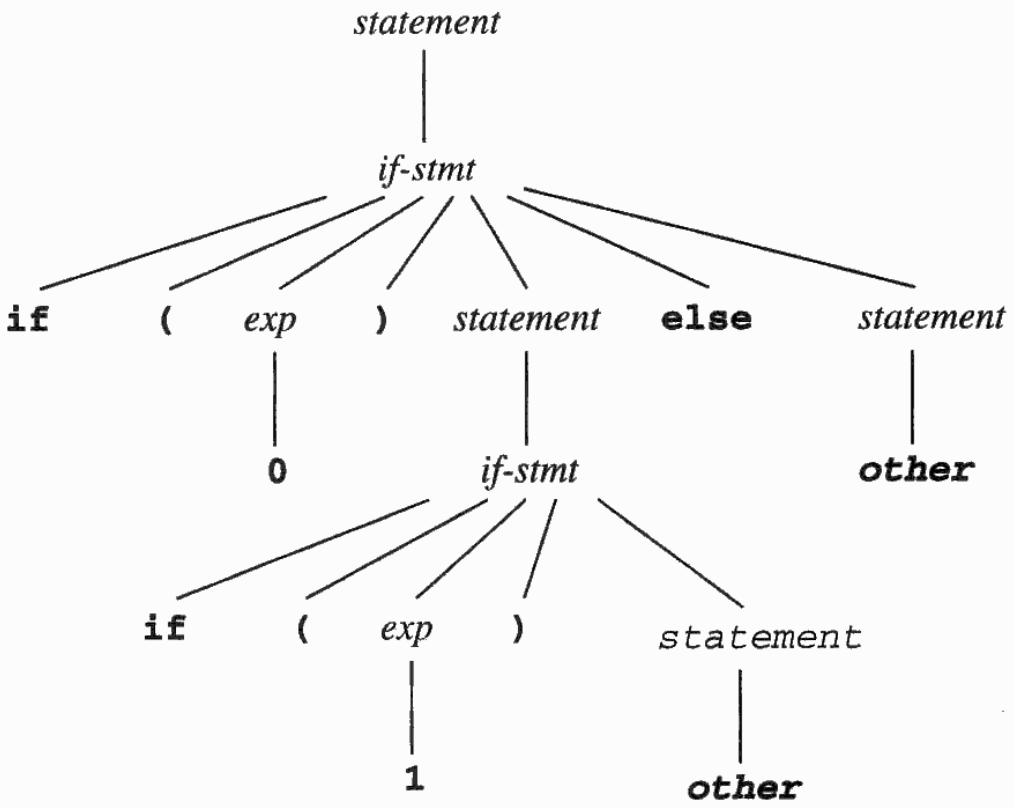
```
if (0) if (1) other else other
```

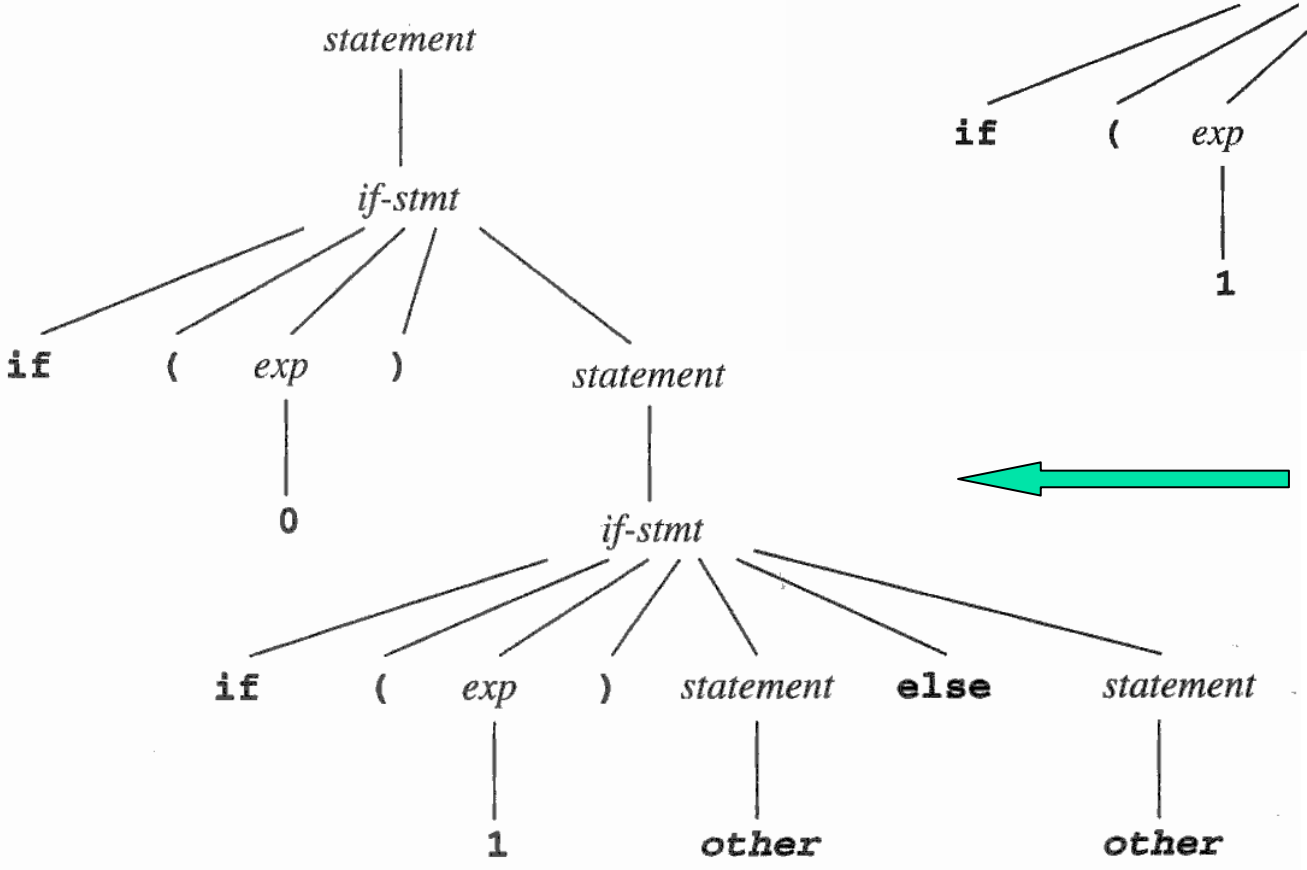
- Grammatikken (tvetydig):

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

To muligheter:

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





Vanlig regel gir denne:

La else bli koblet til nærmeste ”ledige” if

Eks: Entydig grammatikk for if-setning. Gir ”vanlig” løsning

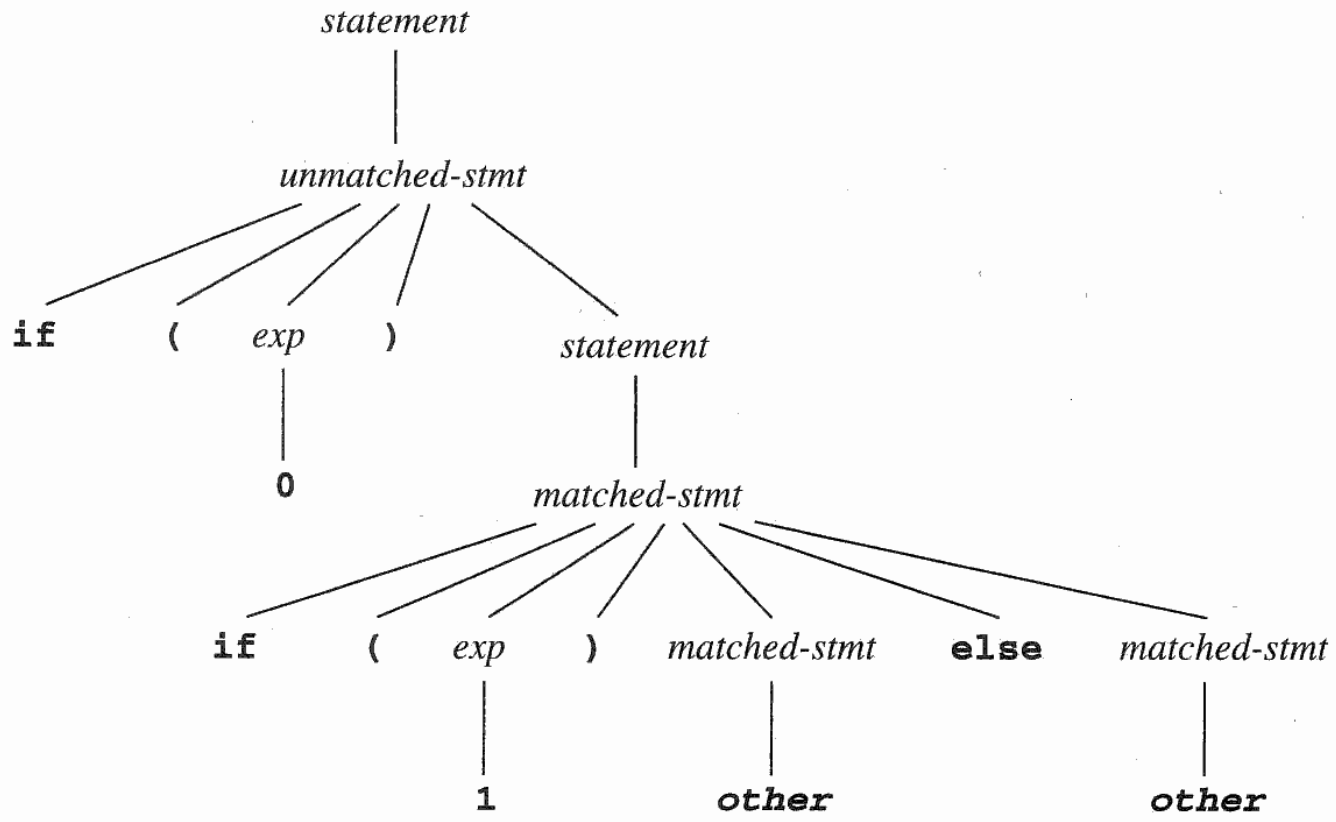
```
if (0) if (1) other else other
```

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:

matched-stmt
- kan ***ikke*** kobles med etterfølgende else

unmatched-stmt
- kan kobles med etterfølgende else

Er det opplagt at denne kan generere alle ”lovlige” setninger (ut fra den kortere flertydige grammatikken på forrige to foiler)?

Utvidet BNF (EBNF)

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

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

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: $A \rightarrow \{\alpha\}\ \beta$

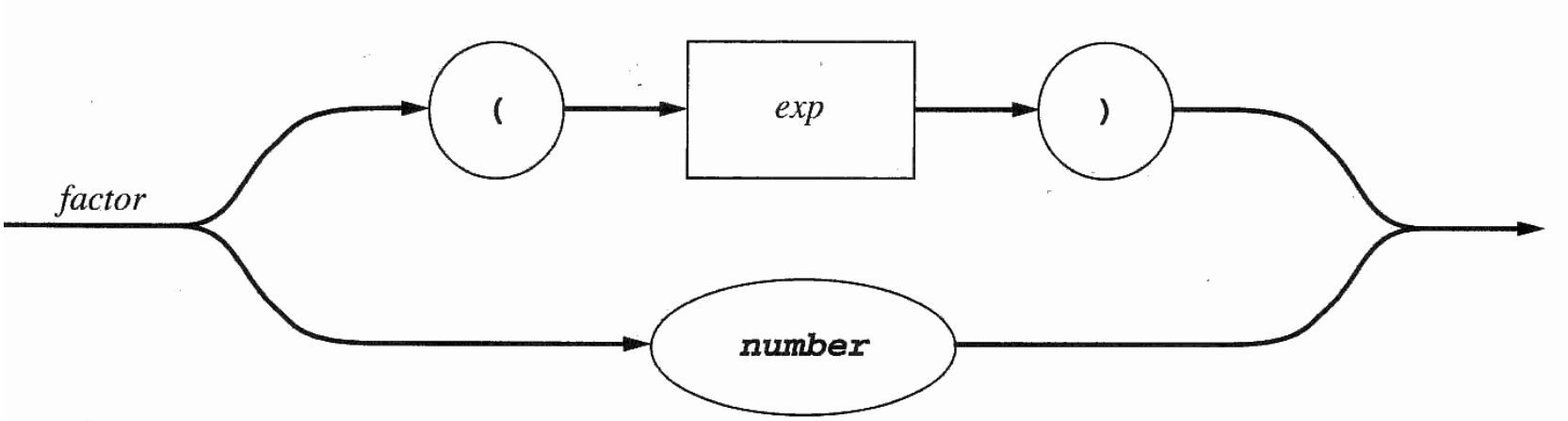
stm-seq → stm {; stm} eller → {stm;} stm

if-setn → if (expr) stm [else stmt]

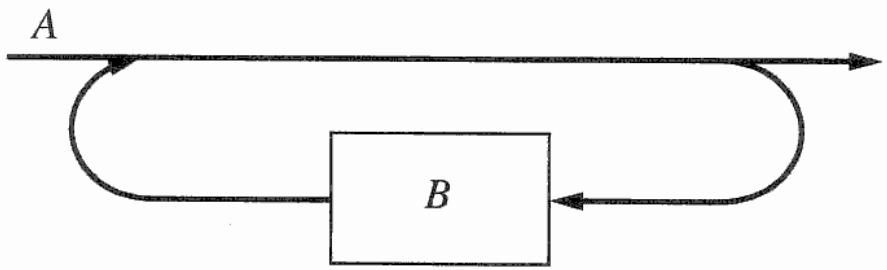
Merk: For en del metoder etc. må man gå ut fra basal BNF.

Syntaks-diagrammer

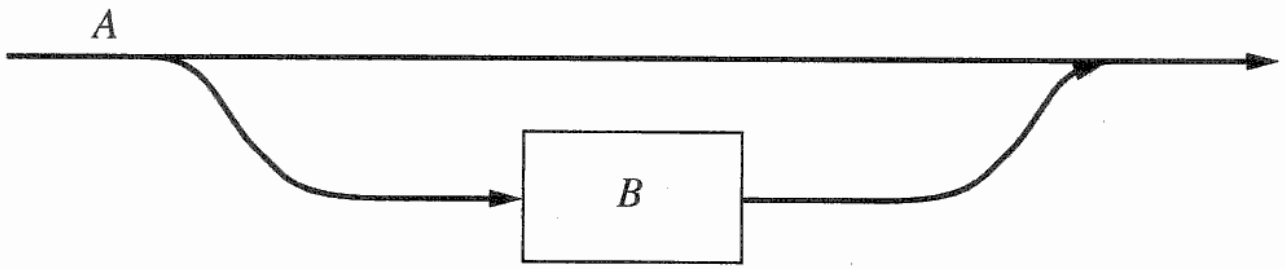
$$factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{number}$$



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



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

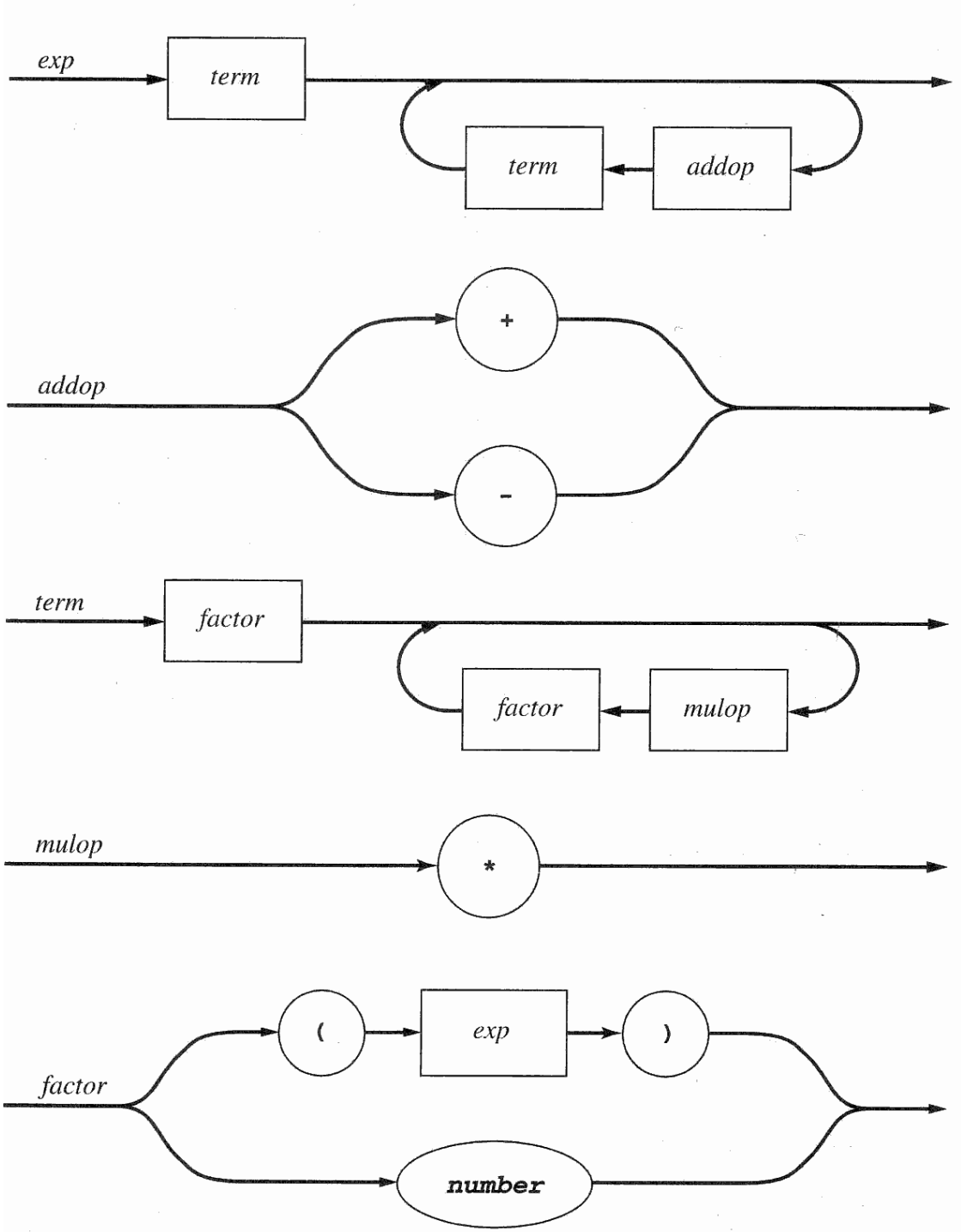


BNF-grammatikk

exp → *exp addop term* | *term*
addop → **+** | **-**
term → *term mulop factor* | *factor*
mulop → *****
factor → **(** *exp* **)** | **number**

EBNF-grammatikk for samme ”språket”, brukes til å lage syntaksdiagrammene

exp → *term* { *addop term* }
addop → **+** | **-**
term → *factor* { *mulop factor* }
mulop → *****
factor → **(** *exp* **)** | **number**



Chomsky-hierarkiet

a er vilkårlig terminalsymb.
β, α, γ er vilkårlig samling av terminal- og ikke-terminalsymb.
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
  - Kontkstfrie prod: $A \rightarrow \alpha$
- Type 3 språk
  - Regulære språk:
    - Regulære utrykk
    - NFA
    - DFA

Produksjoner bare på formen:
$A \rightarrow Ba$ *og* $A \rightarrow a$
eller
$A \rightarrow aB$ *og* $A \rightarrow a$

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 TINY

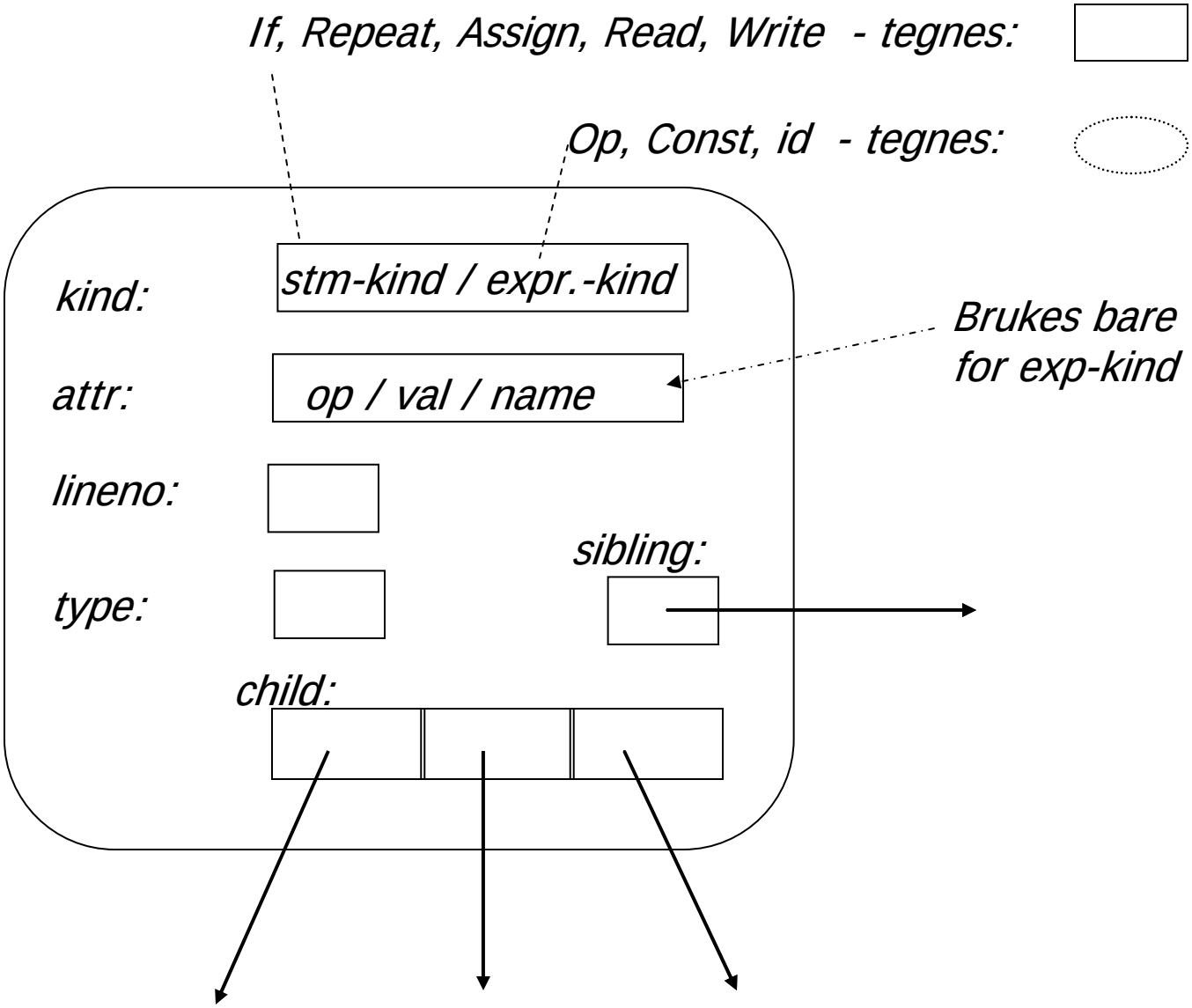
```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK}
             StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* 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 /sublklasser **som implementasjons-språk.**

Syntaks-tre for et program i Tiny

```
{ Sample program
  in TINY language–
  computes factorial
}
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
```

```
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
```