INF 5110, 25. januar 2013

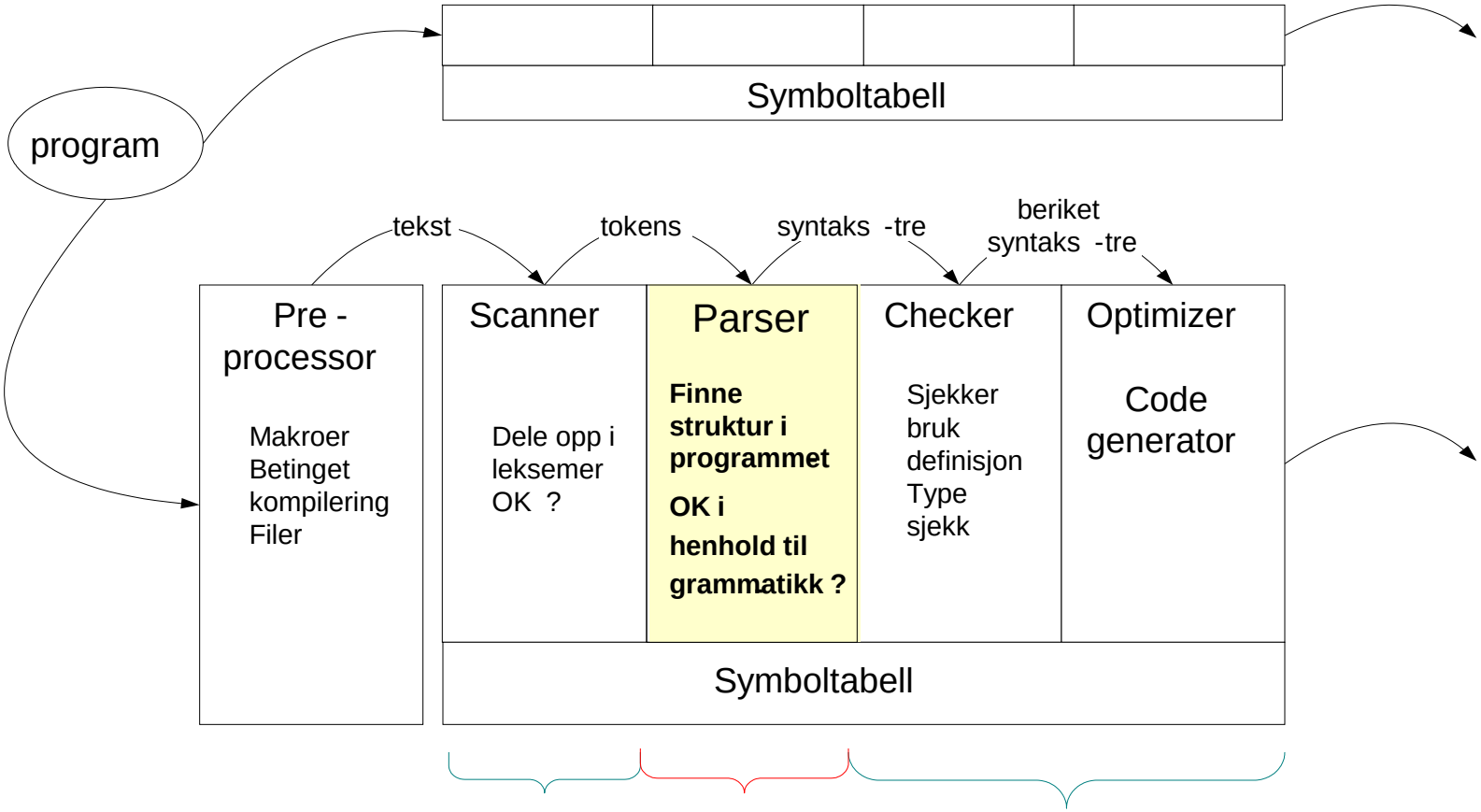
Stein Krogdahl

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

Hvor er vi nå - kap. 3, 4 og 5:

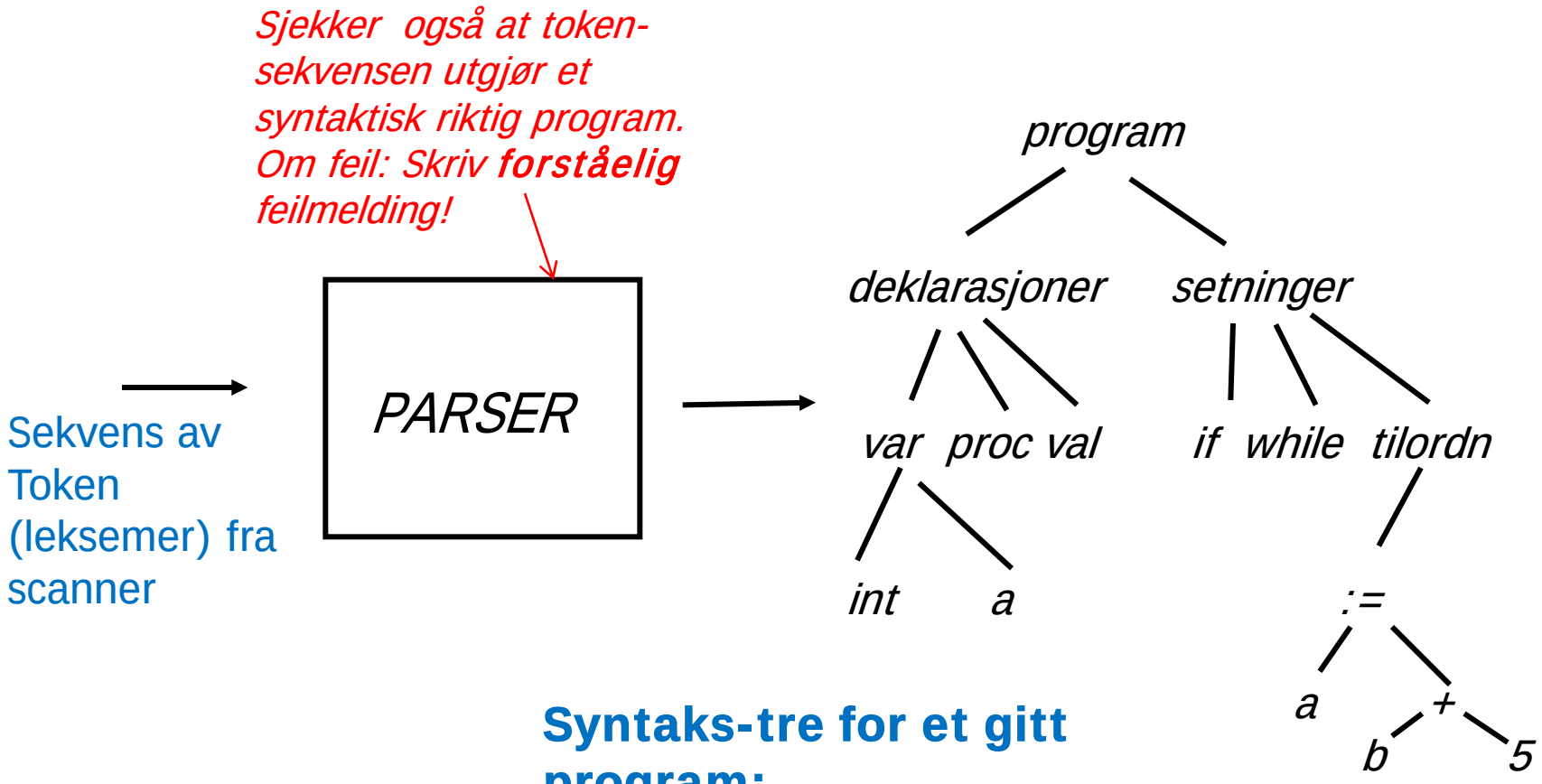


Lex
Flex

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

Attributtgrammatikker
+
div. mer eller mindre systematiske metoder

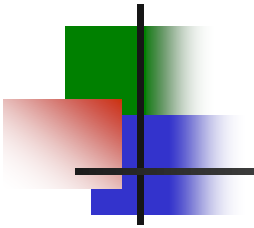
Forenklet skisse av hva en parser gjør



Syntaks-tre for et gitt program:

”Abstrakt” eller ”konkret”?

Dette treet er typisk **abstrakt**.



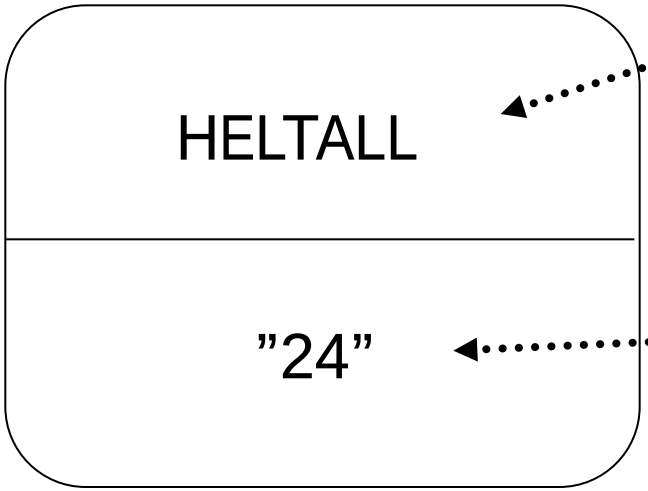
Oversikt – kap 3 (grunnleggende om grammatikker)

- 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

- Og: Før vi starter på kapittel 4 tar vi også en del generelt stoff om grammatikker. Dette er i boka er vevd innimellom i kap. 4 og 5, men vi tar det samlet før disse kapittelene.

Hva får vi fra scanneren

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



Token (eller ”token-klasse”):
”typen” av symbolet:
- *navn, heltall, arit-op,* ”if”

Leksem: Hvordan symbolet så ut i programmet, eller en kode som angir hvilket leksem det er innen token-klassen)

- 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 → 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):

Ikke-terminal

Metasymbol

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

Metasymbol *Terminal-symbol*

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

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

Flere måter å skrive den samme grammatikken

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

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

6 regler eller "produksjoner"

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

$$
\begin{aligned}
E &\longrightarrow E\ O\ E \mid (\ E\ ) \mid \mathbf{n} \\
O &\longrightarrow + \mid - \mid *
\end{aligned}
$$

Avledning (her *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

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

grammatikk *streng med bare terminal-symboler*

Avledning (her 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 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)

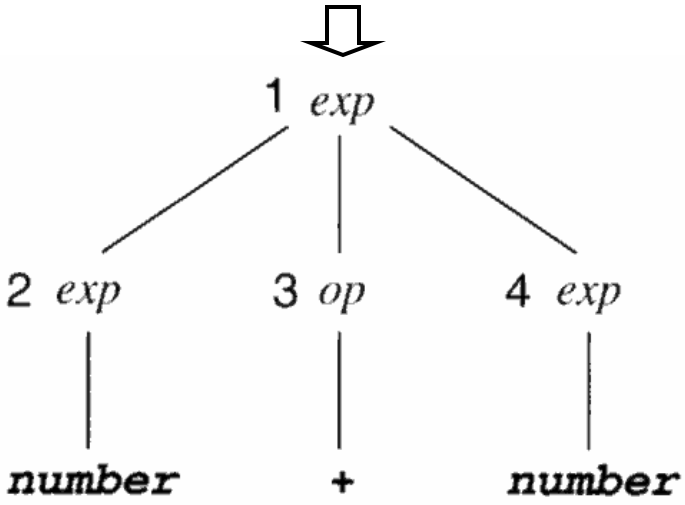
(1) $exp \Rightarrow exp\ op\ exp$

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

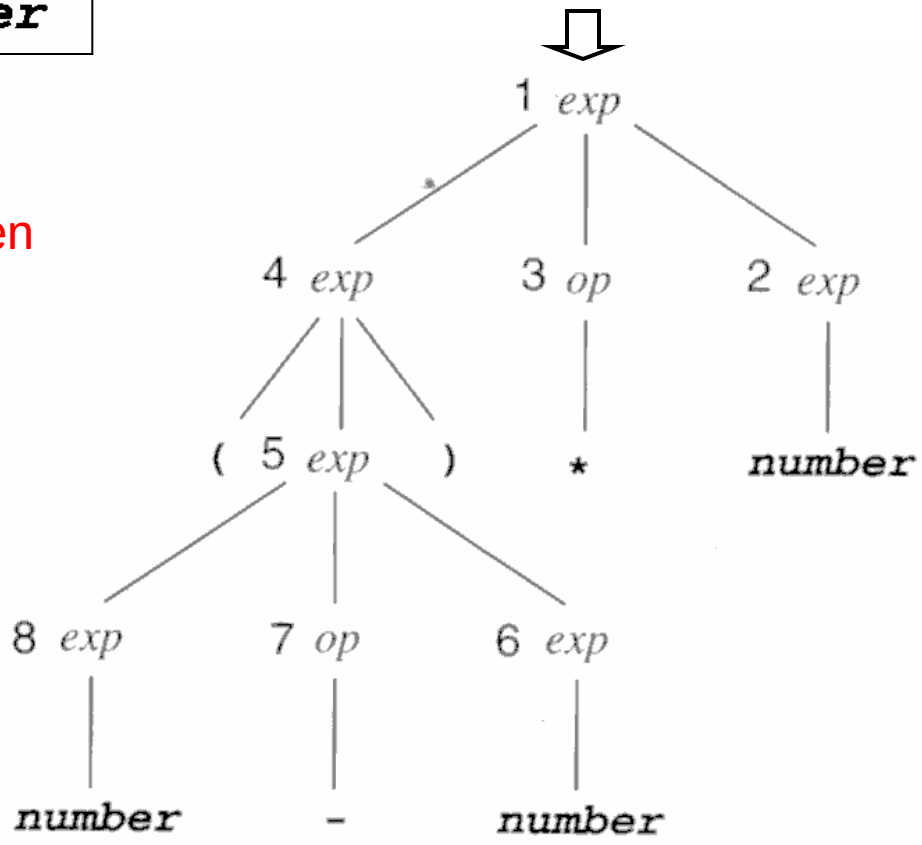
(3) $\Rightarrow$ **number** + exp

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

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

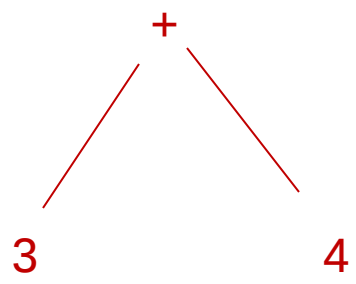


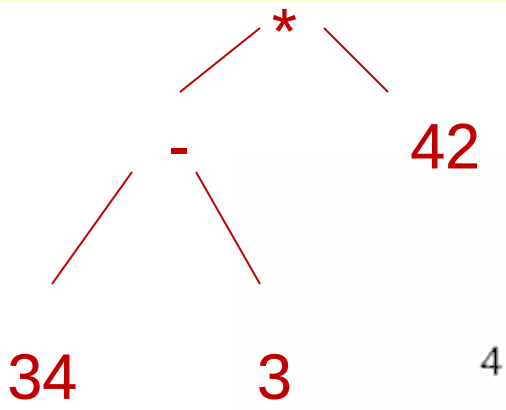
- Exp: **(num – num) * num**
- Tallene angir høyre-avledning
- Ser vi bort fra tallene, gir altså alle avlednings-rekkefølger det samme treet:

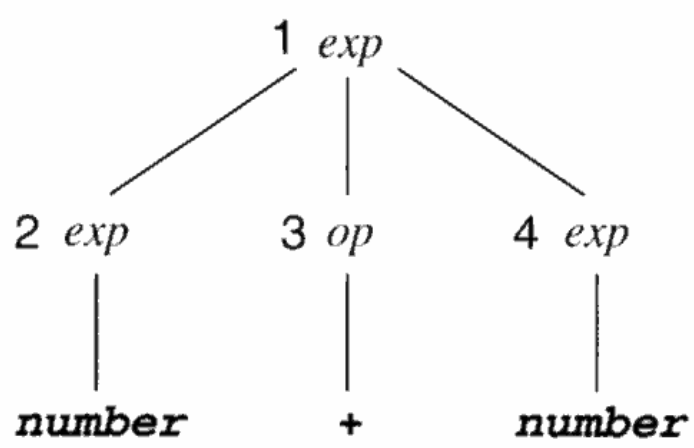


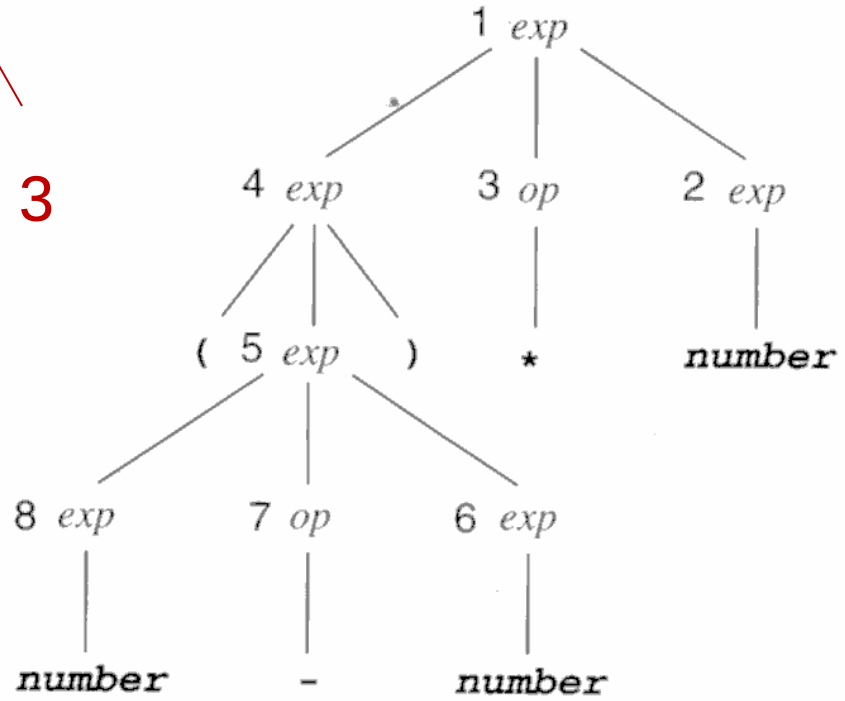
Abstrakt syntakstre (AST)

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.

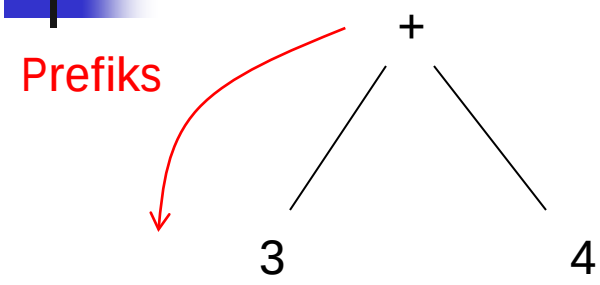


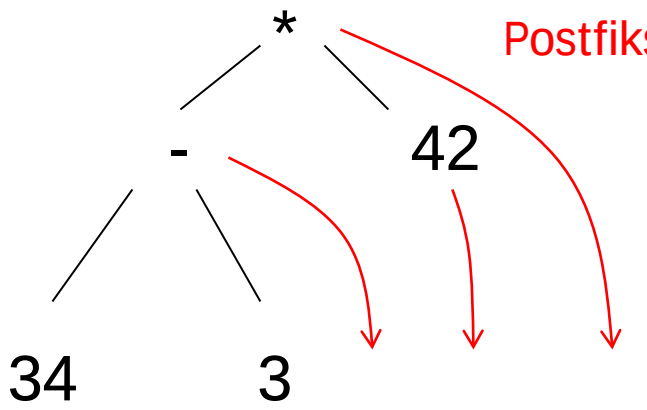






Abstrakt syntakstre





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

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 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*

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

Merk1: Om det er *et kjent antall operander* for hver operator så vil:

Postfiks og prefiks gi et entydig tre

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

Merk2: Postfiks egner seg for beregning med operand-stakk

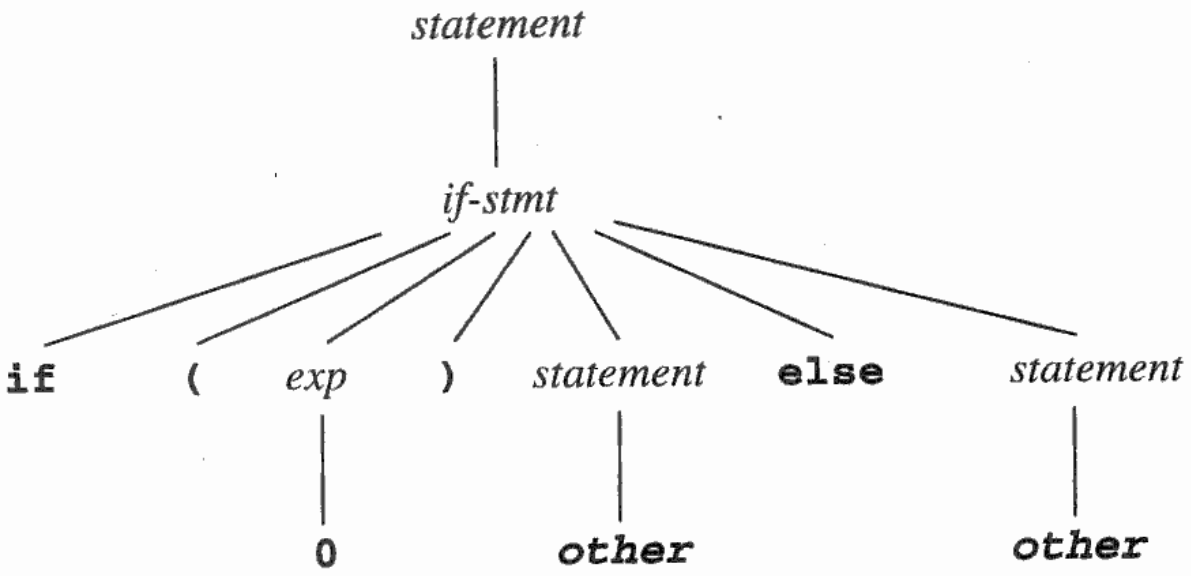
Flere parserings-trær (konkrete sytaks-trær)

G1:

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

Merkelig betingelse

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

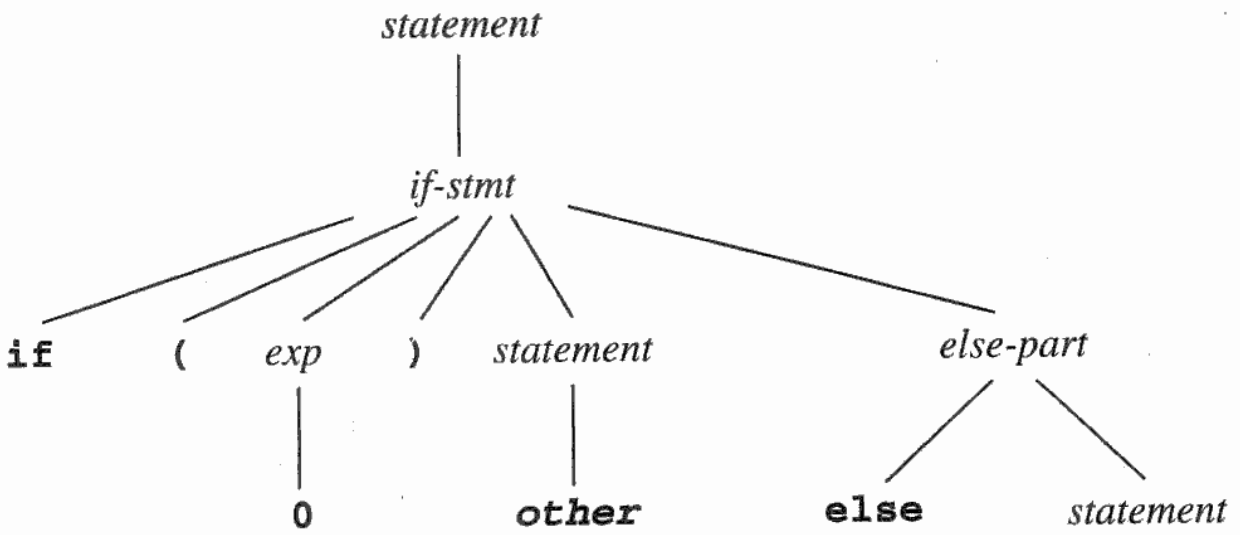


En annen grammatikk G2 for if-setninger

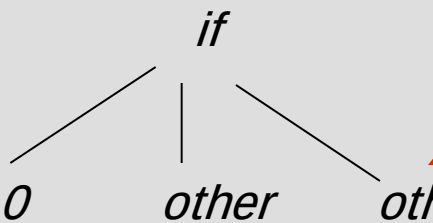
G2:

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

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)*

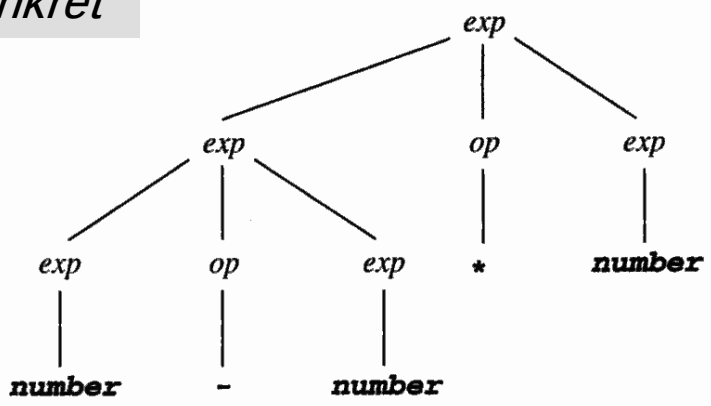
Flertydige grammatikker
Analyse av setningen: n - n * n

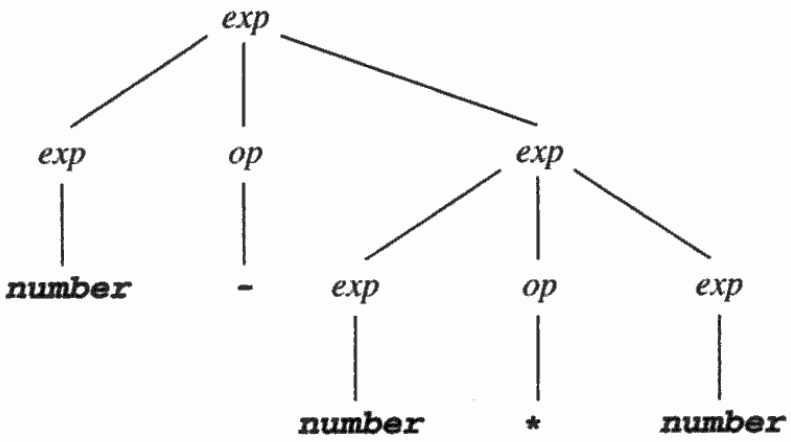
G er *flertydig* hvis det finnes en setning i L(G) som kan gis **flere** parserings-trær

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

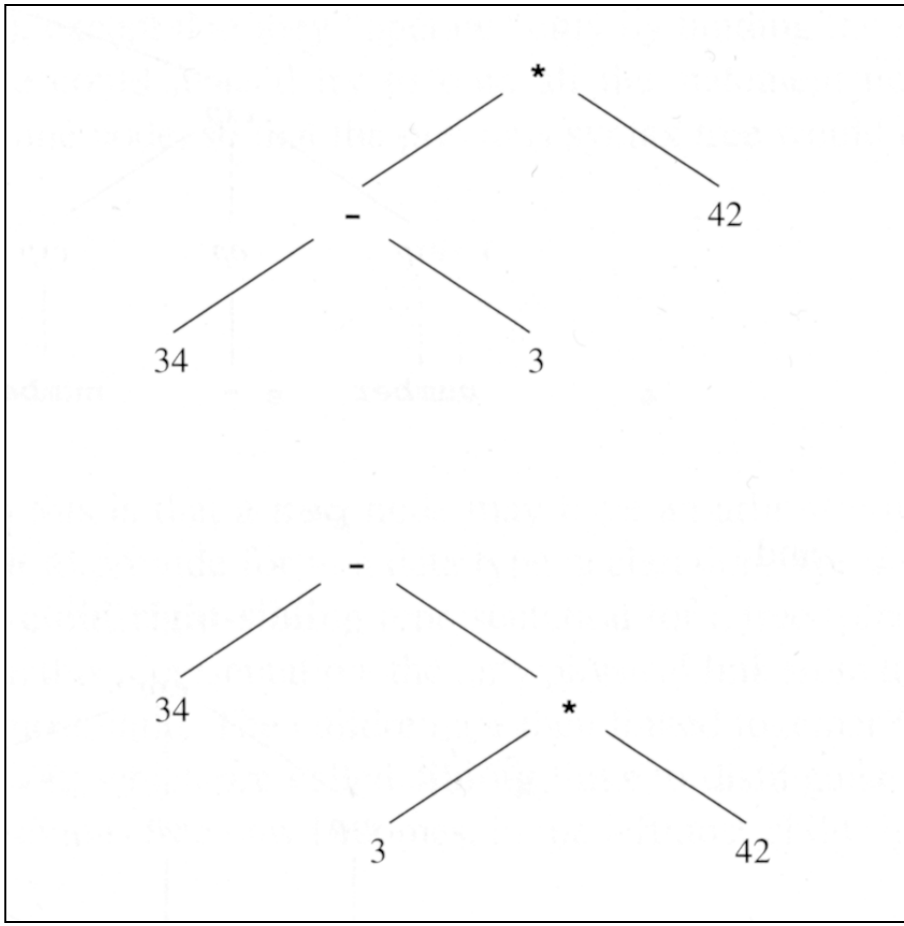
$$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \textbf{number}$$
$$op \rightarrow + \mid - \mid *$$

Konkret





Abstrakt



Bruk av presedens og assosiativitet for å gjøre flertydige grammatikker entydige.

- 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 bare *ett* lovlig 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

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

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

- 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" :

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

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

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

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

To entydige grammatikker

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

$addop \rightarrow$ **+** $\mid$ **-**

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

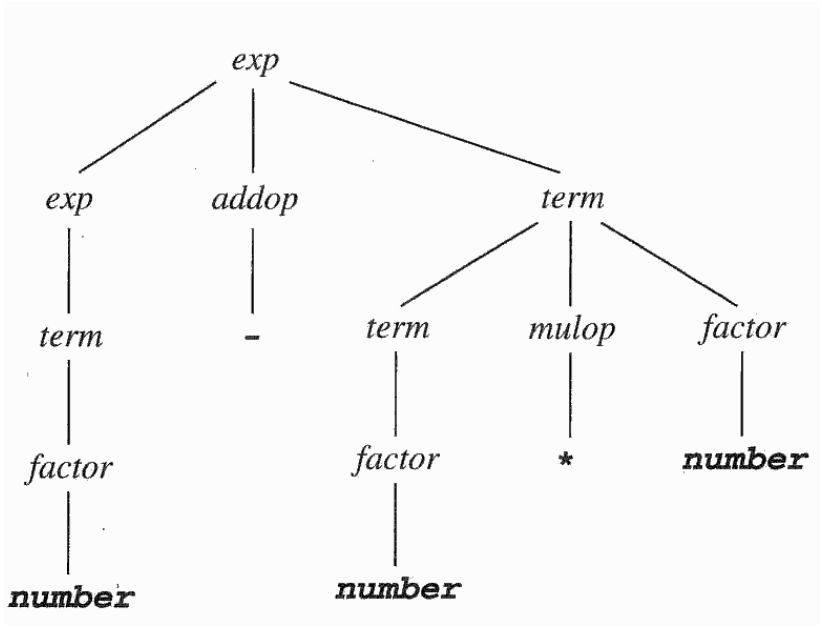
$mulop \rightarrow$ ***

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

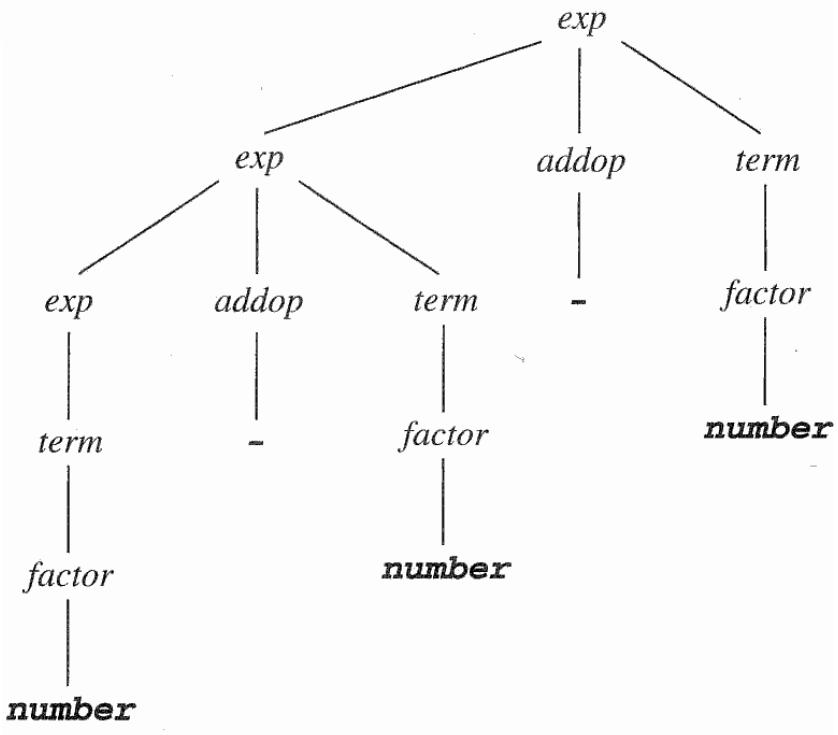
Rekkefølgen angir venstre/høyre assosiativitet

En ekstra ikke-terminal for hvert presedens-nivå

34-3*42



34-3-42



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

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):

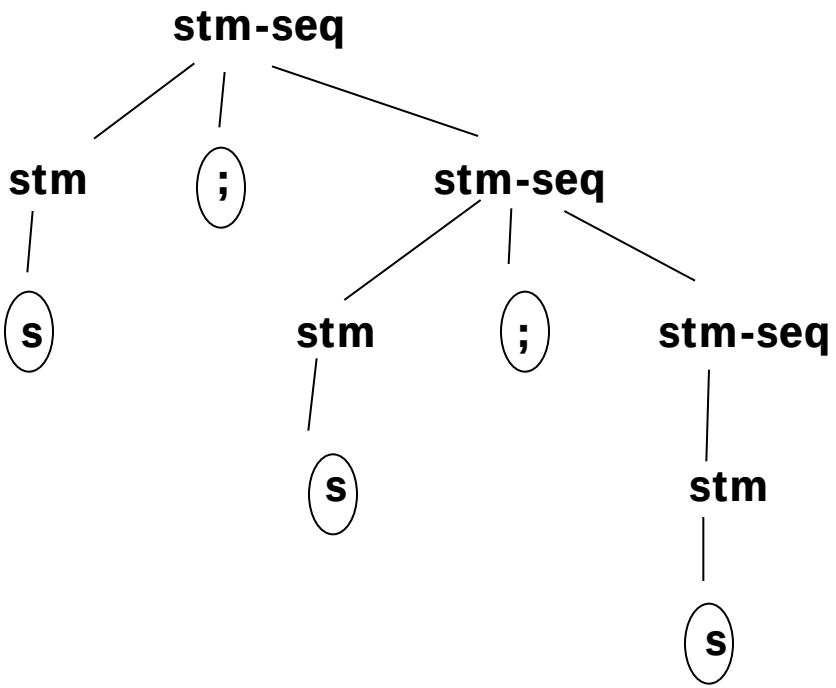
postfix ops	[] . (⟨exp⟩) ⟨exp⟩ ++ ⟨exp⟩ --
prefix ops	++⟨exp⟩ --⟨exp⟩ -⟨exp⟩ ~⟨exp⟩ !⟨exp⟩
creation/cast	**new** (⟨type⟩)⟨exp⟩
mult./div.	* / %
add./subt.	+ -
shift	<< >> >>>
comparison	< <= > >= **instanceof**
equality	== !=
bitwise-and	&
bitwise-xor	^
bitwise-or	|
and	&&
or	||
conditional	⟨bool_exp⟩?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

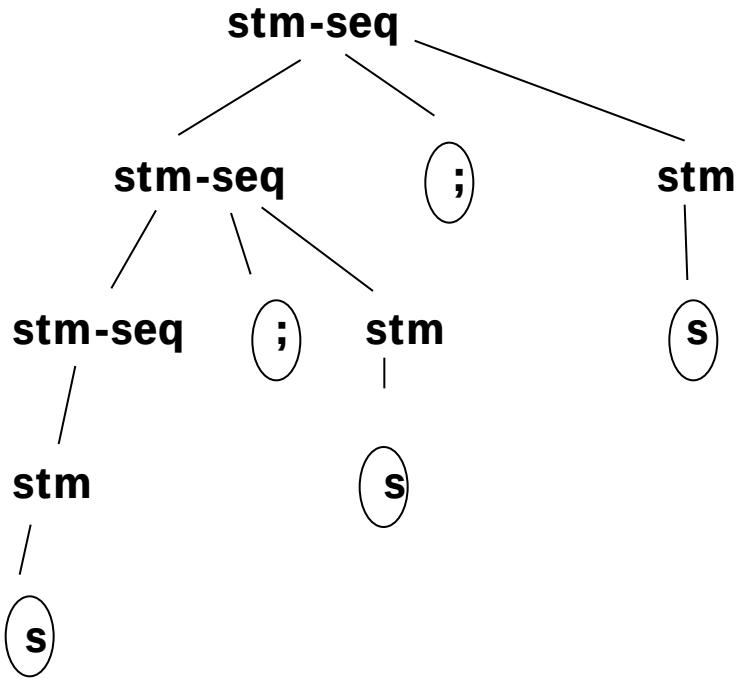
Ikke-essensiell flertydighet

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

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

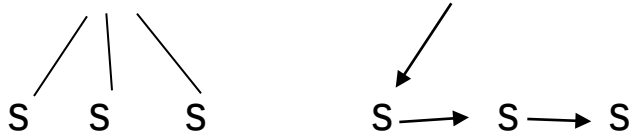
stm → s





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*

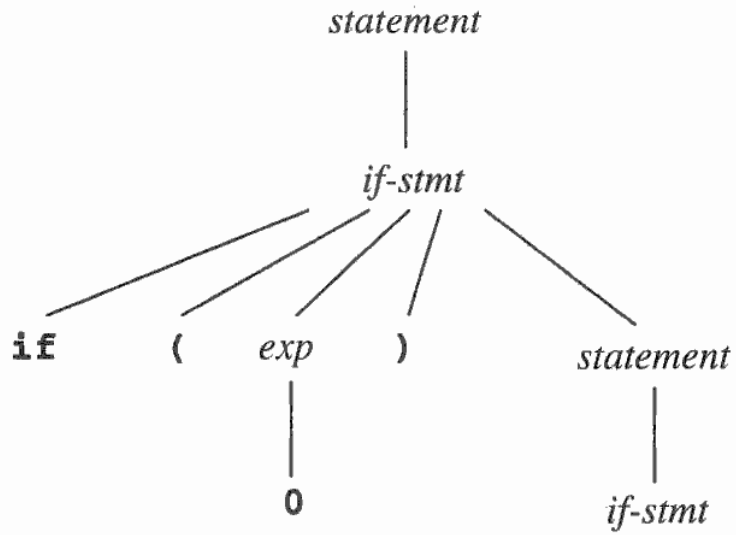
exp → **0** | **1**

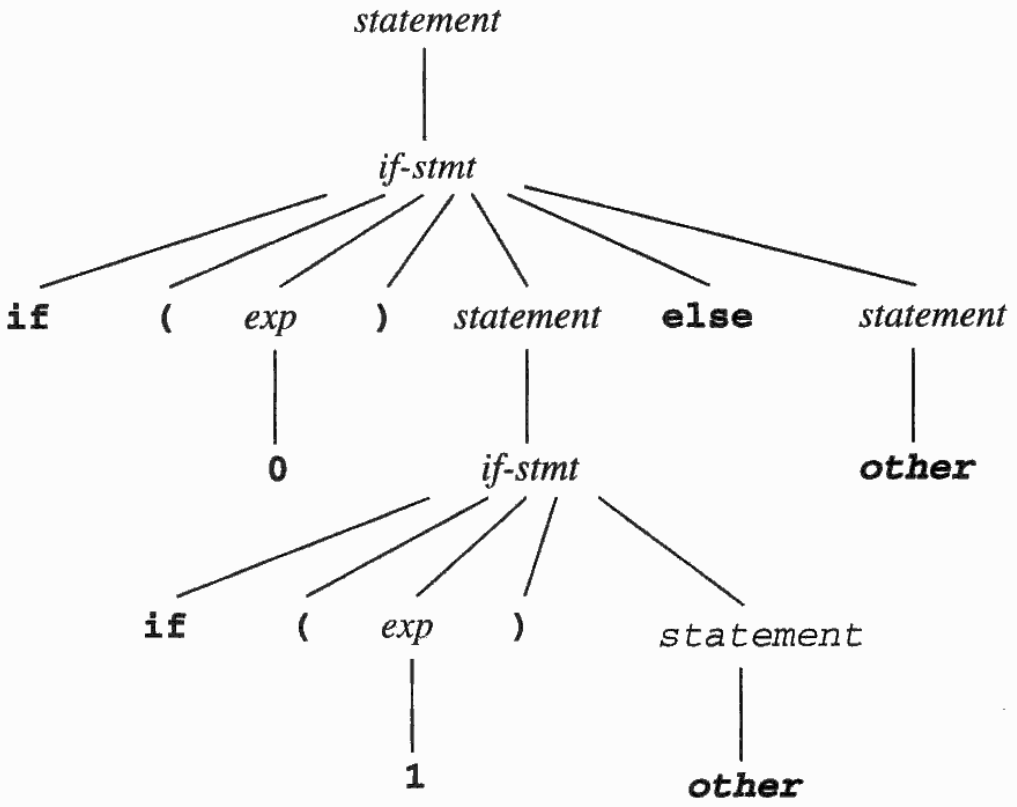
To trær for samme setning

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**

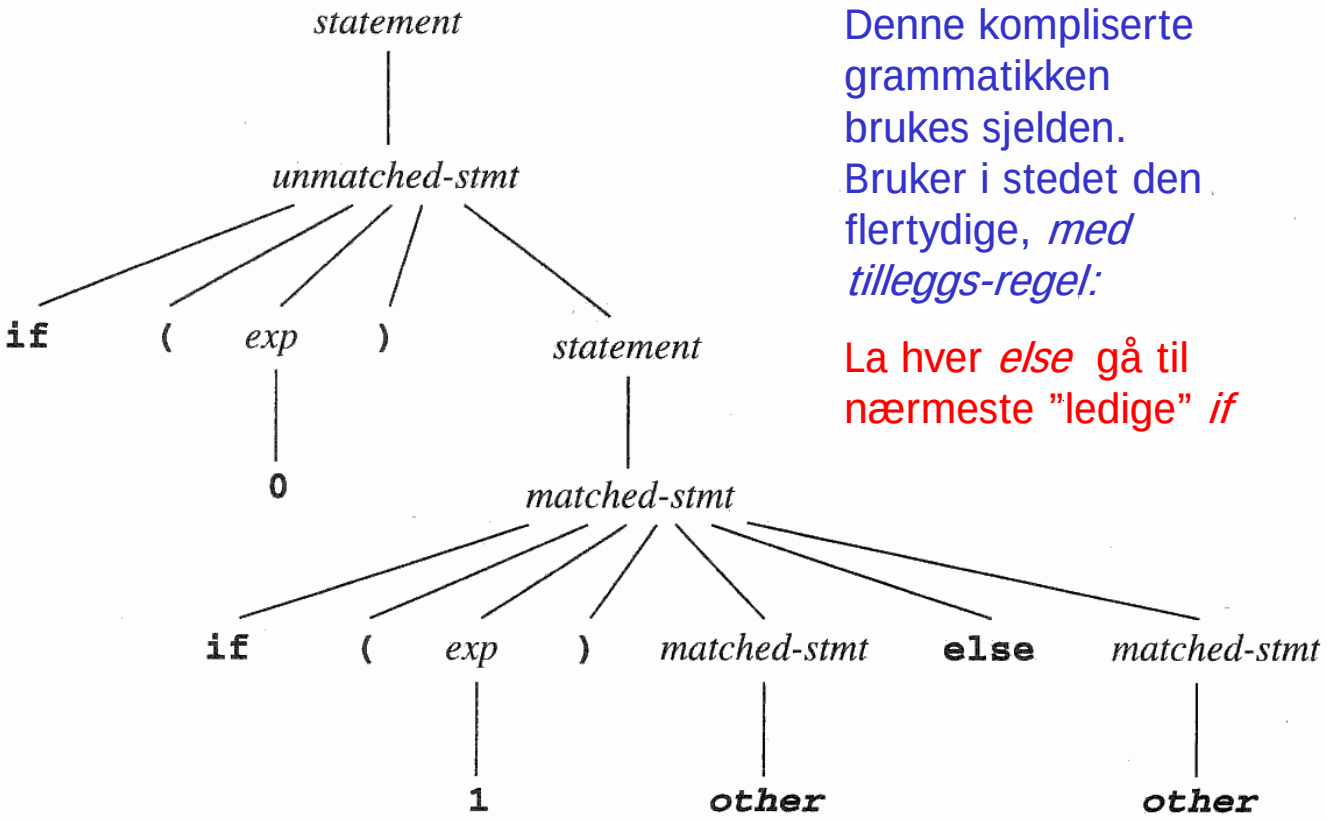
Idé:

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 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: $\{\alpha\}$ 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

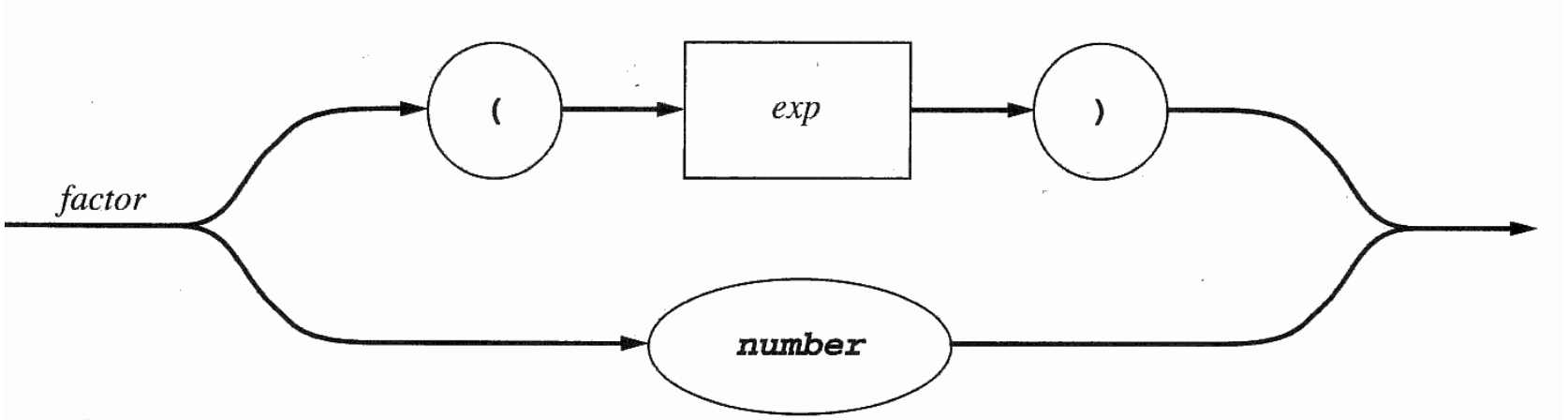
if-setn → if (expr) stm [else stmt]

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

Syntaks-diagrammer

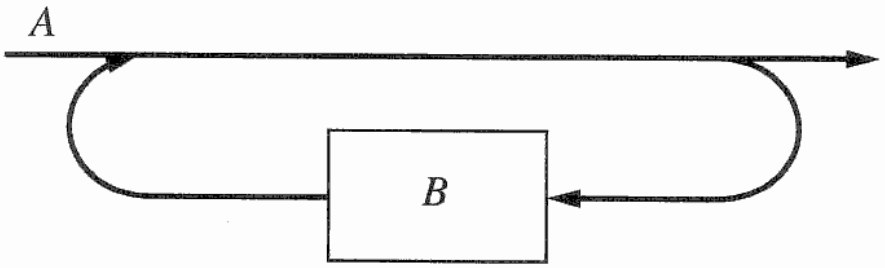
Omtrent som ikke-deterministiske automater for regulære språk, men her kan de være «rekursive» (direkte eller indirekte).

factor → (*exp*) | **number**

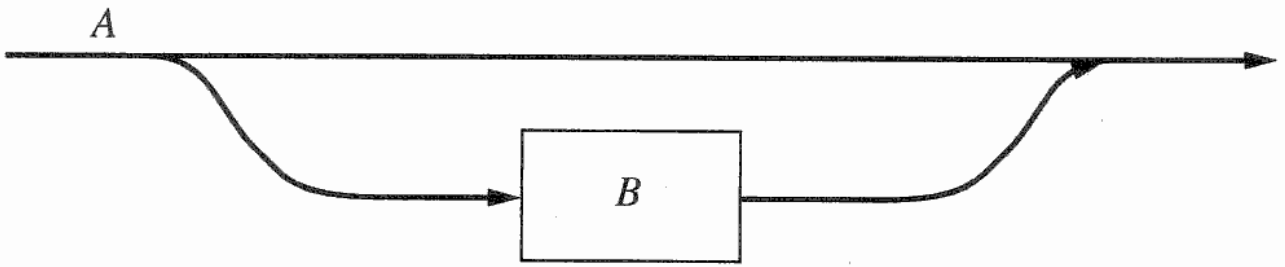


Merk: *"Entydige syntakstre"* og andre liknende begreper er ikke så naturlig å definere her

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



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



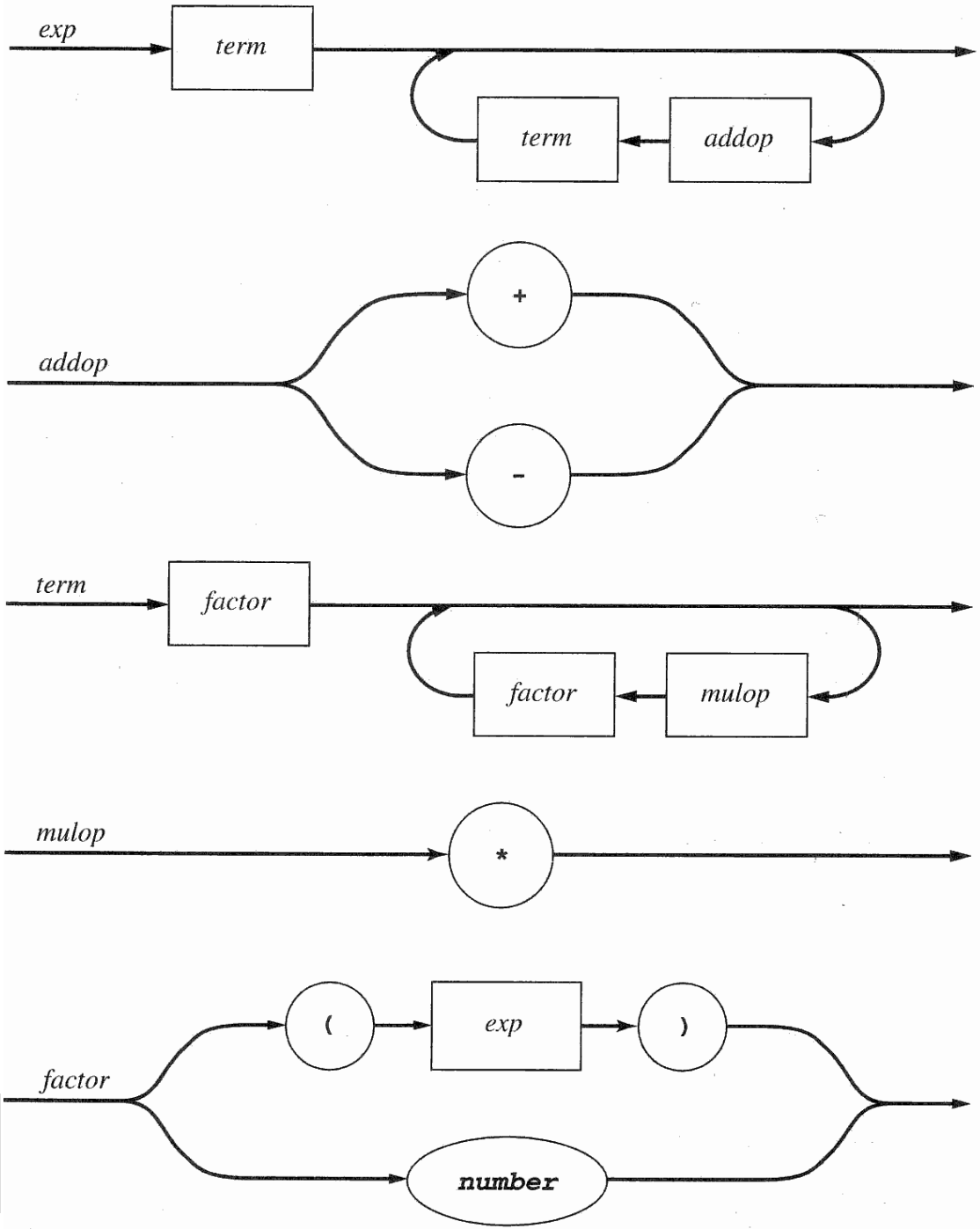
BNF-grammatikk:

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

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

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

Merk: Her må *assosiativitet* (men ikke presedens) 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)

Tilsvarer: Turingmaskiner

- **Type 1 – språk**
  - Kontekst-sensitive produksjoner:

$\beta A \gamma \rightarrow \beta \alpha \gamma$

Tilsvarer :"Lineæet begrensede automater"

- **Type 2 – språk** (det vi bruker til vanlig syntaks-beskrivelse)
  - Kontkstfrie prod., (E)BNF:

$A \rightarrow \alpha$

Tilsvarer: Automat med stakk-lager

- **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* $A \rightarrow a$

Tilsvarer: Endelige automater

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 tilording
- 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 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

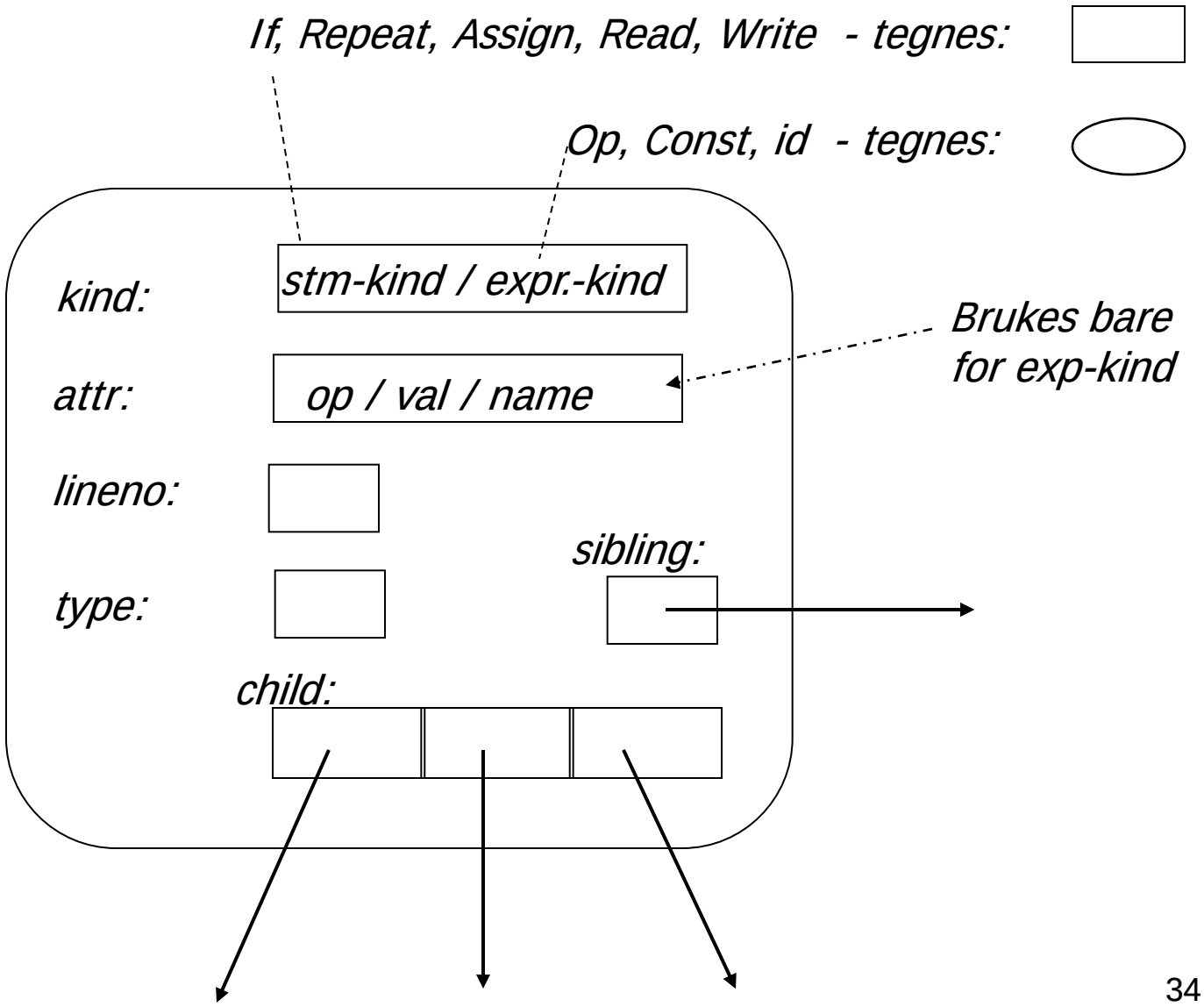
```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK}
             StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* 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



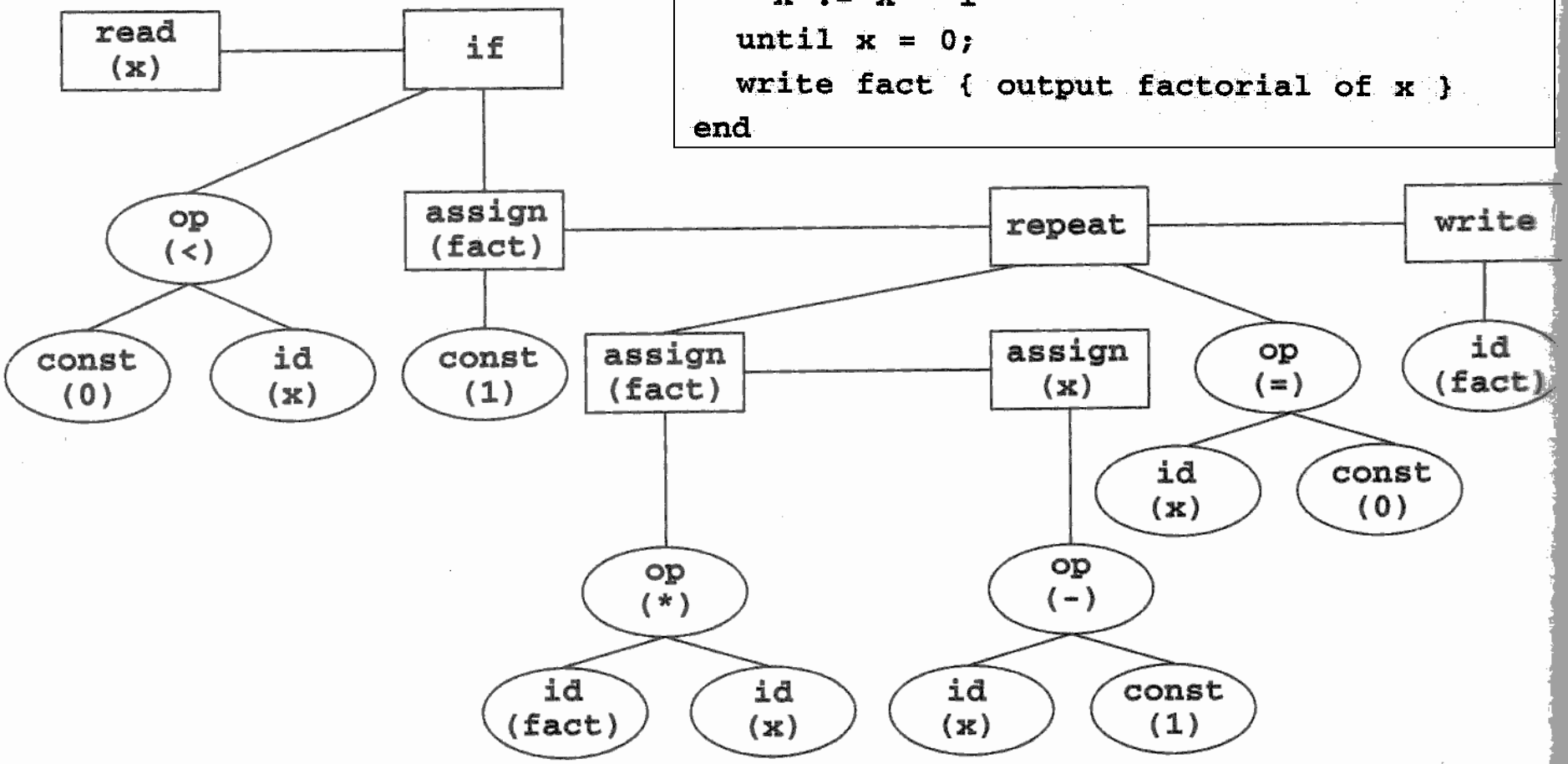
Denne node-strukturen kan uttrykkes bedre om vi bruker et OO-språk som implementasjons-språk, der vi har klasser og sublklasser.

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

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?*