

INF5110, 15. februar 2011

Hovedstoffet i kap 4:  
Parsering ovenfra-ned ("top down")

Stein Krogdahl Ifi, UiO

**Forbedret "endelig" utgave, lagt ut 16/2**

**Oppgaver som gjennomgås i dag:**

- Spørsmålene på de to siste foilene fra tirsdag 8/2 (Bl.a. Lag et subklassehierarki som ville passe for nodene i et abstrakt syntakstre for Tiny)
- Finn *First*- og *Follow*-mengdene til grammatikken i eksempelet på side 160 *etter at* venstre-rekursjon er fjernet (se også foil 16 fra 9/2)
- Beskriv en algoritme som bare finner de utnullbare ikke-terminaler



# 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
```

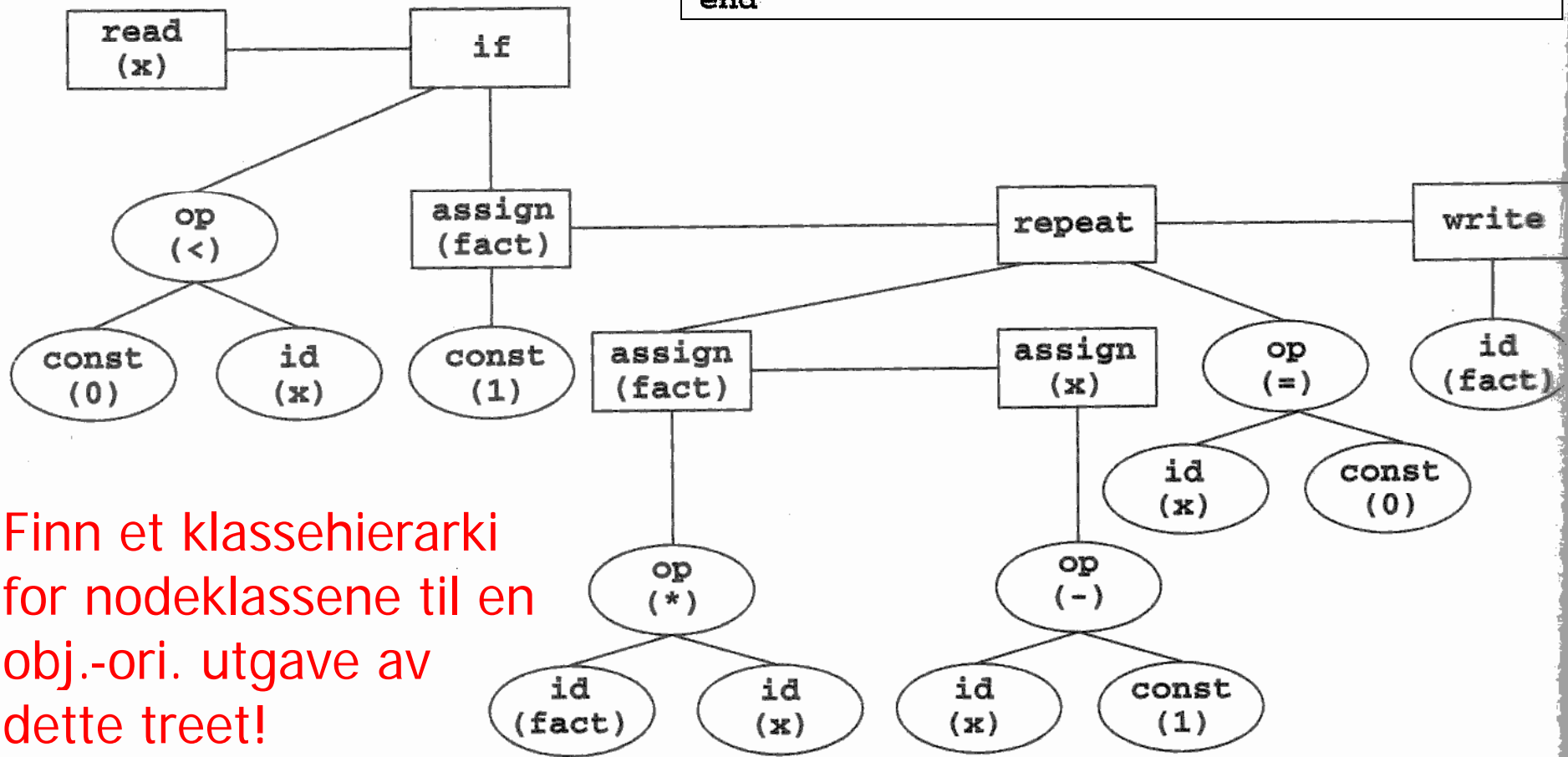
- Er grammatikken entydig?
- Hva om vi vil tillate tomme setninger
- Hva om vi vill ha semikolon etter og ikke mellom setningene?
- Hva slags assosiativitet og presedens er det for operatorene?

# Svar på spørsmålene på forrige foil

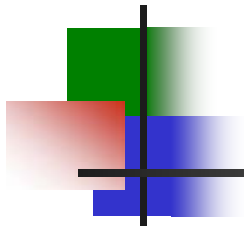
- Er grammatikken entydig?
  - Dette er generelt *uavgjørbart* for generelle BNF-grammatikker
  - Vi skal se på metode for å avgjøre det for mange praktiske grammatikker
  - Denne er ihvertfall delt opp i presedens-nivåer, og har assosiativites-angivelse, og er nok derved entydig. If-setninger med både **else** og **end** er ikke noe problem for entydigheten (*her sa jeg feil på forelesningen!*).
- Hva om vi vil tillate tomme setninger
  - Det er bare å sette til et tomt alternativ for *statement*
  - Den ser ut til fremdeles å være entydig
- Hva om vi vil ha semikolon etter og ikke mellom setningene?
  - Bytt ut reglen for *stmt-sequence* med:  
*stmt-sequence* -  $\rightarrow$  *stmt-sequence statement ; | statement ;*
- Hva slags assosiativitet og presedens er det for operatorene?
  - Høyest presedens    \* /        Venstre-assosiativ
  - Midlere presedens    + -        Venstre-assosiativ
  - Lavest presedens     < =       Ikke-assosiativitet (bare to operander)
  - Kunne altså brukt flertydig grammatikk, med disse tilleggs-reglene

```
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
```

# Abstrakt syntakstre for Tiny-programmet:



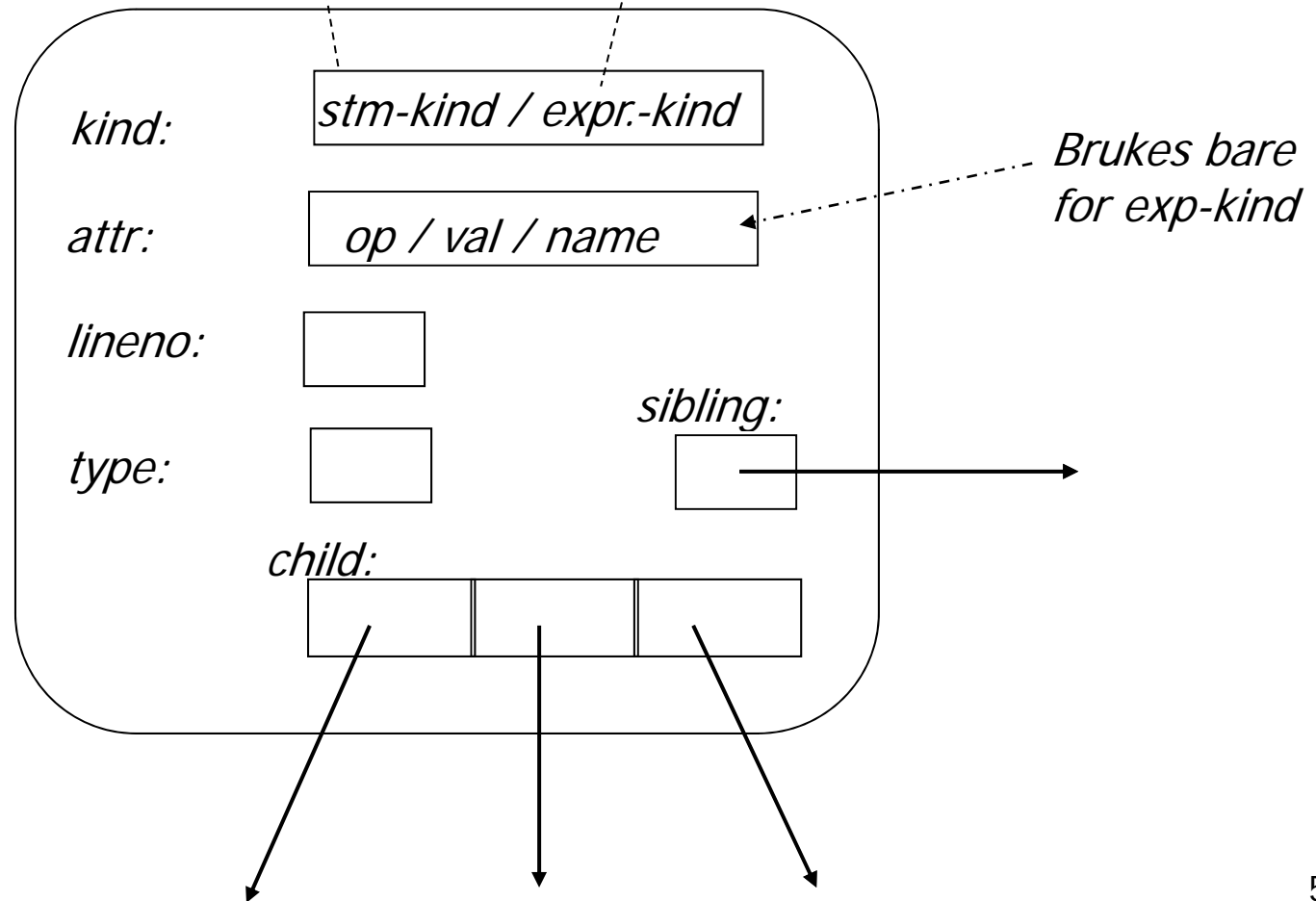
Finn et klassehierarki for nodeklassene til en obj.-ori. utgave av dette treet!



# Nodestruktur i C for Tiny

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

*Op, Const, id* - tegnes:

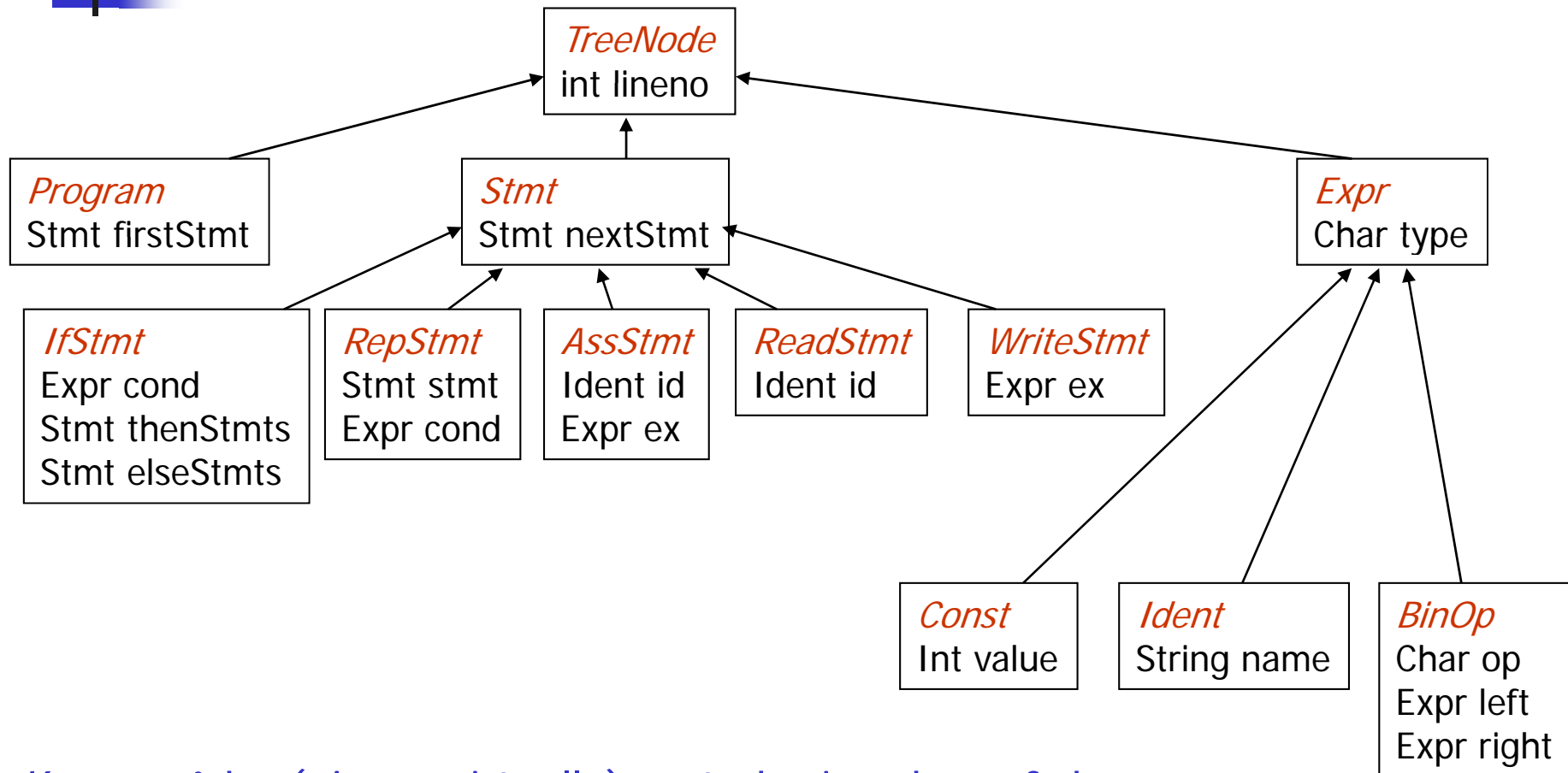


*Brukes bare for expr-kind*

Denne nodestrukturen passer enda bedre med et OO-språk med klasser /subklasser som **implementasjonsspråk**.

## Nodeklasser for OO-utgave av abst.-synt.-tre for Tiny-språket

Merk: Dette er altså en fast subklasse-struktur i kompilatoren, og må ikke forveksles med det abstrakte syntaks-treet for et gitt program



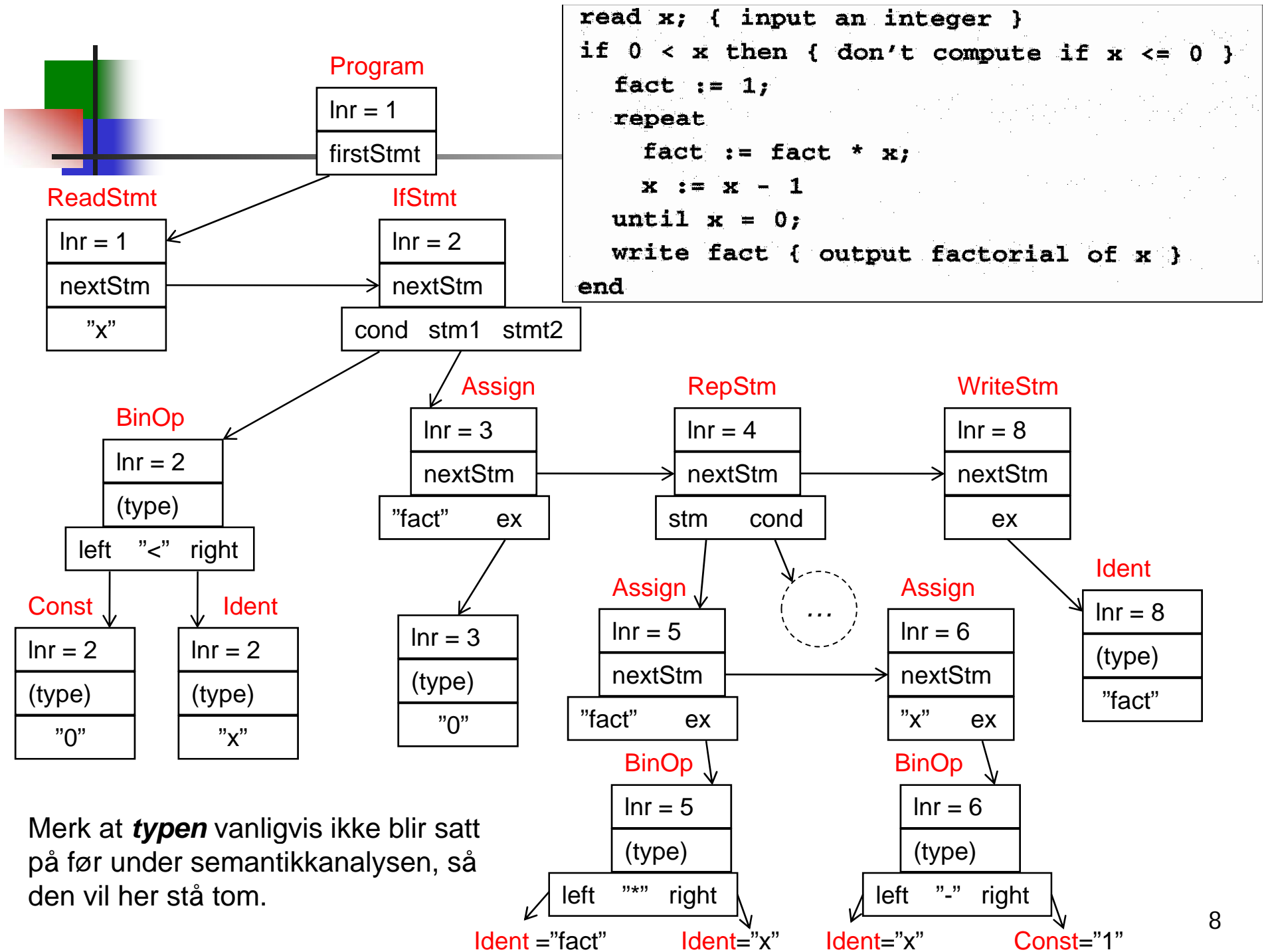
Kan også ha ( gjerne virtuelle) metoder i nodene, f.eks:

- `doSemAnalyses()`; Gjør semantisk analyse av noden, og av subtreet det er rot i
- `generateCode()`; Genererer kode for noden, og for subtreet det er rot i



# Tiny-grammatikken

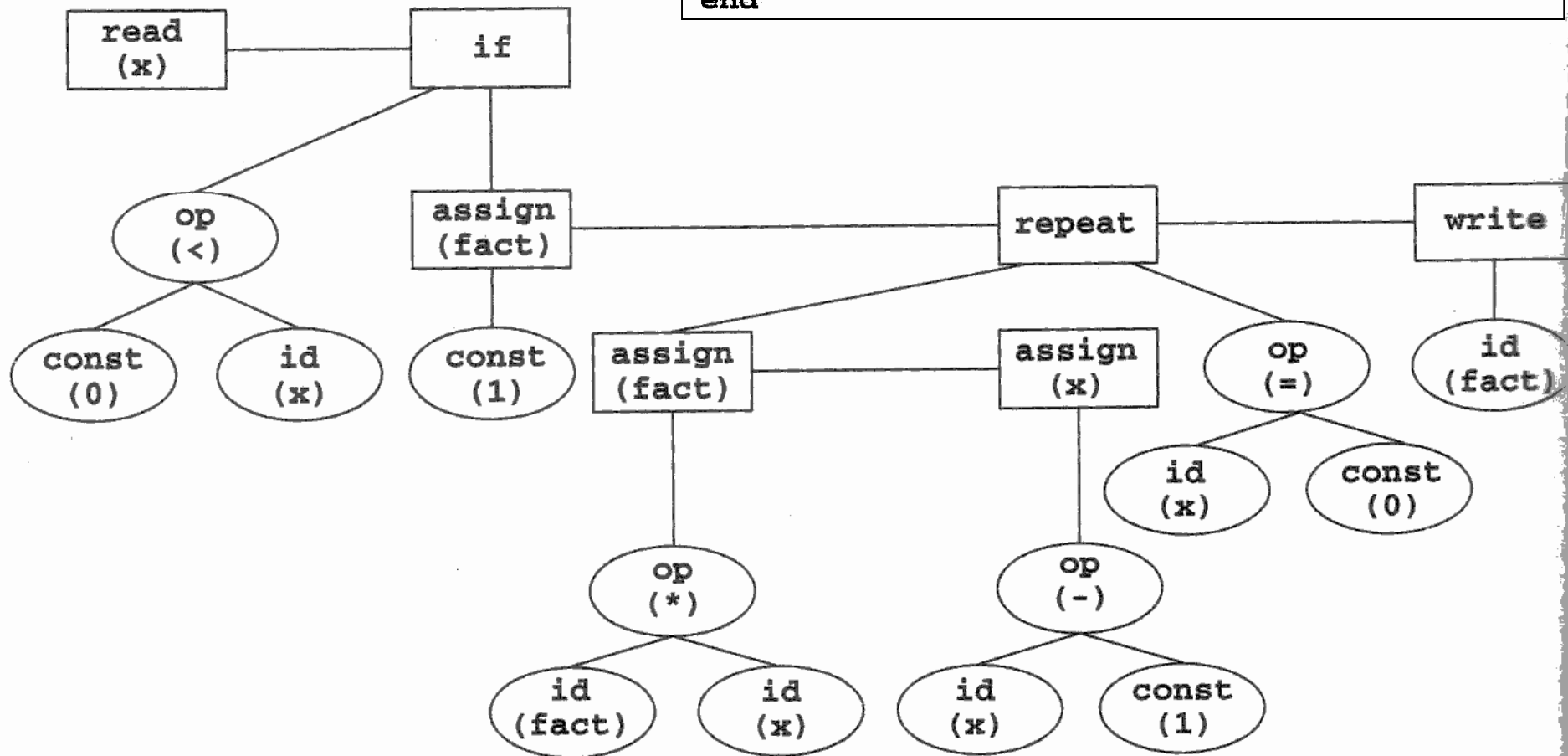
*program* → *stmt-sequence*  
*stmt-sequence* → *stmt-sequence* ; *statement* | *statement*  
*statement* → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*  
*if-stmt* → **if** *exp* **then** *stmt-sequence* **end**  
          | **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **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**





## Abstrakt syntakstre for Tiny-programmet:

```
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
```



# First og follow for transformert uttr.gram.

Har fjernet venstre-  
rekursjon:

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

Vi får da følgende First- og  
Follow-mengder:

$$\text{First}(\text{exp}) = \{ (, \mathbf{number} \}$$
$$\text{First}(\text{exp}') = \{ +, -, \varepsilon \}$$
$$\text{First}(\text{addop}) = \{ +, - \}$$
$$\text{First}(\text{term}) = \{ (, \mathbf{number} \}$$
$$\text{First}(\text{term}') = \{ *, \varepsilon \}$$
$$\text{First}(\text{mulop}) = \{ * \}$$
$$\text{First}(\text{factor}) = \{ (, \mathbf{number} \}$$
$$\text{Follow}(\text{exp}) = \{ \$, ) \}$$
$$\text{Follow}(\text{exp}') = \{ \$, ) \}$$
$$\text{Follow}(\text{addop}) = \{ (, \mathbf{number} \}$$
$$\text{Follow}(\text{term}) = \{ \$, ), +, - \}$$
$$\text{Follow}(\text{term}') = \{ \$, ), +, - \}$$
$$\text{Follow}(\text{mulop}) = \{ (, \mathbf{number} \}$$
$$\text{Follow}(\text{factor}) = \{ \$, ), +, -, * \}$$



## Algoritme for å beregne mengden av utnullbare ikke-terminaler (kan gjøres først)

---

Vi skal ha en mengde "UNB" av ikke-terminaler:

- Er fra starten tom
- Skal få nye elementer etter en regel 1 og 2 under
- Når den ikke øker mer er vi ferdig

1. Først legges alle ikke-terminaler  $A$  som har en produksjon  $A \rightarrow \epsilon$ , inn i UNB

2. (Gjentas til det ikke blir mer forandring):

Om en ikke-terminal  $A$  har en produksjon  $A \rightarrow B \dots D$  der alle  $B \dots D$  er ikke-terminaler som allerede er med i UNB, så legg også  $A$  inn i UNB.



# Kap. 4: Parsering ovenfra-ned (top-down)

---

Dette bør leses om igjen etter kapittelet:

- *First* og *Follow*-mengder
  - Boka tar det et stykke uti kap 4, vi tok det først (forrige gang)
- *LL(1)-parsering* og boka og pensum:
  - Det som i *boka* kalles LL(1)-parsering (4.2.1, 4.2.2 og 4.2.4) er en metode for top-down parsering med en eksplisitt stakk. Dette er *ikke* med som pensum.
  - Vi konsentrerer oss i dette kapittelet om "recursive descent"-parsering, en intuitiv metode som mange sikkert har vært litt borti (INF2100, ++)
  - Ofte brukes betegnelsen LL(1)-parsering også om "rec. desc."-metoden brukt ut fra syntaksdiagrammer, EBNF eller ren BNF, men man har da gjerne et ikke-teknisk forhold til om ting helt sikkert fungerer riktig.
  - Vi skal se på det tekniske kravet for at en ren BNF-grammatikk kan parseres rett fram med "rec. desc."-metoden.
  - **LL(1)-grammatikk i vår betydning:**  
Det er en ren BNF-grammatikk som tilfredstiller kravet fra forrige punkt

# Eksempel: Rec. desc. av ren BNF

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

Neste token  $\swarrow$   $\nwarrow$  Global variabel

if a + b \* ( c + d ) <= ...

## Hoved-idé:

- Skriv en funksjon/prosedyre/metode for hver ikke-terminal
- La denne finne **riktig alternativ (helst fra bare førstkommande token)**, og gjør parsing av den videre input ut fra det

"Typisk" rec.decent-prosedyre for siste produksjon over, som blir veldig enkel.

```
procedure factor ;
begin
  case token of
    ( : match( ) ;
      exp ;
      match( ) ) ;
    number :
      match(number) ;
  else error ;
  end case ;
end factor ;
```

Sjekker at angitt terminal kommer, og "leser til neste". Brukes ofte *bare* for å lese (sjekken må slå til). Da er det egentlig nok å kalle "getToken" (men her kalles alltid "match")

```
procedure match ( expectedToken ) ;
begin
  if token = expectedToken then
    getToken ;
  else
    error ;
  end if ;
end match ;
```

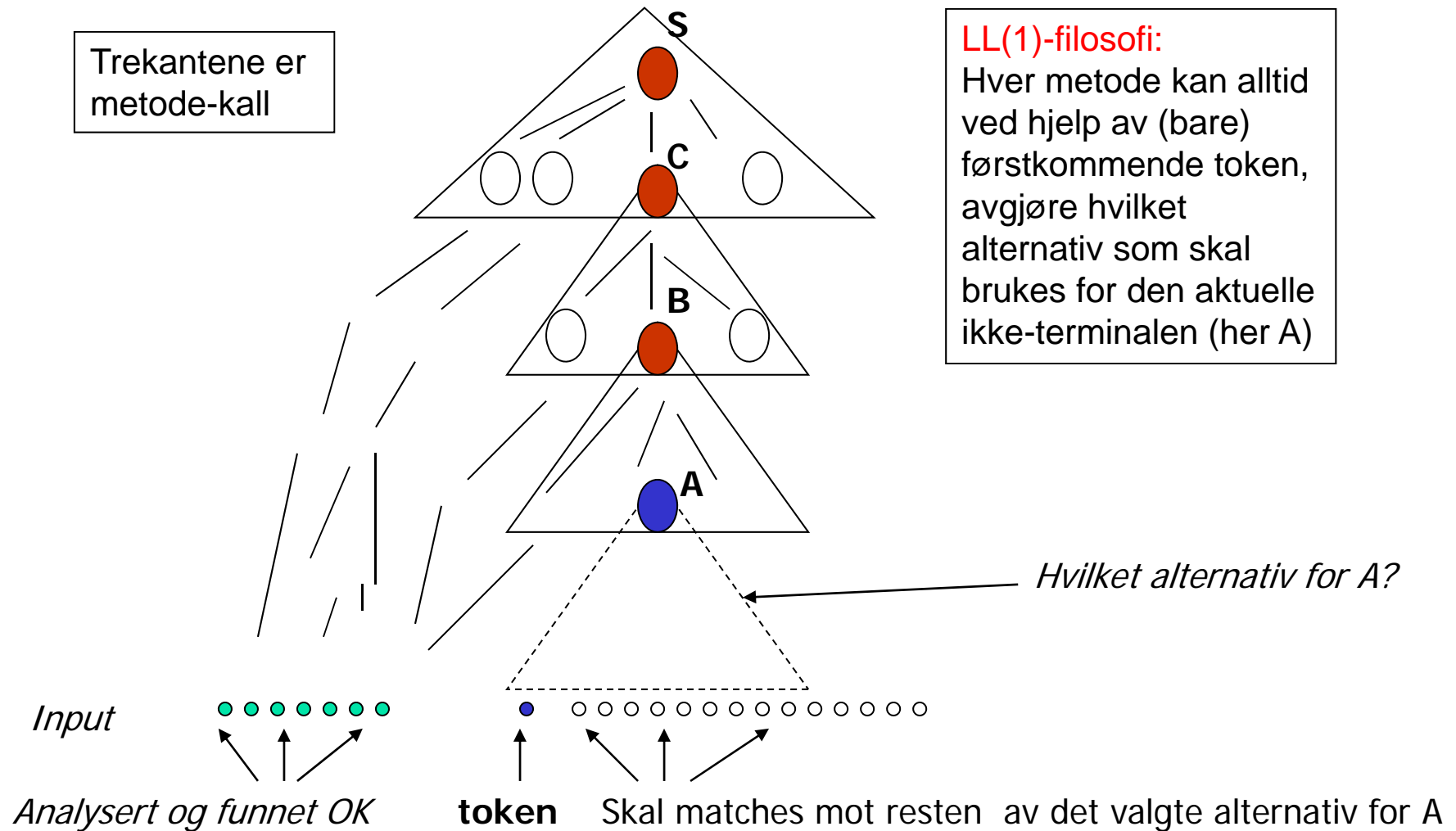
# Situasjonen under rekursiv parsering

Kalles altså "top down"-parsering (ovenfra-ned-parsering)

Trekantene er metode-kall

## LL(1)-filosofi:

Hver metode kan alltid ved hjelp av (bare) førstkomende token, avgjøre hvilket alternativ som skal brukes for den aktuelle ikke-terminalen (her A)



Ved litt mer kompliserte tilfeller virker ikke ren BNF bra, men med venstrefaktorisering eller EBNF går det her greit

Opprinnelig

$$\begin{aligned} \text{if-stmt} \rightarrow & \mathbf{if} ( \text{exp} ) \text{ statement} \\ & | \mathbf{if} ( \text{exp} ) \text{ statement} \mathbf{else} \text{ statement} \end{aligned}$$

Skrives ut som:

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

R-D-prosedyre:

```
procedure ifStmt ;
begin
  match (if) ;
  match ( ( ) ;
  exp ;
  match ( ) ;
  statement ;
  if token = else then }
    match (else) ;
    statement ;
  end if ;
end ifStmt ;
```

NB: Kunne også bruke venstre-faktorisering. Da ville dette bli en egen prosedyre "elsePart":

$$\begin{aligned} \text{ifStmt} & \rightarrow \underline{\mathbf{if}} ( \text{exp} ) \text{ stmt} \text{ elsePart} \\ \text{elsePart} & \rightarrow \varepsilon \mid \underline{\mathbf{else}} \text{ stmt} \end{aligned}$$

# Venstre-rekursjon gir problemer ved ren BNF. Men går ofte med EBNF

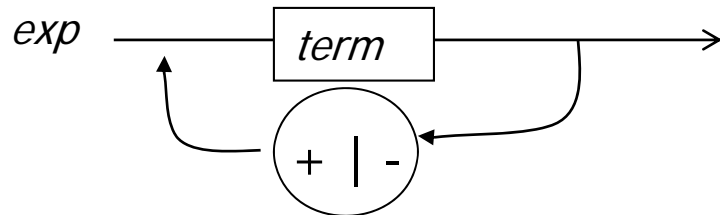
*Gir uendelig mange rekursive kall*

$exp \rightarrow exp \text{ addop } term \mid term$

Bruker EBNF:

$exp \rightarrow term \{ \text{addop } term \}$

```
procedure exp ;
begin
  term ;
  while token = + or token = - do
    match (token) ;
    term ;
  end while ;
end exp ;
```

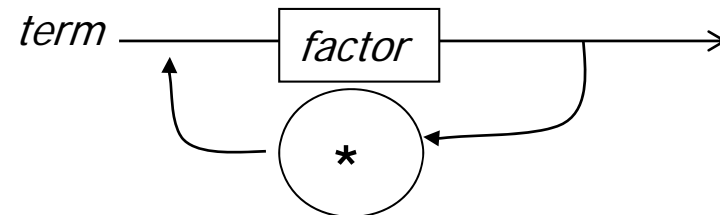


NB: Kan også fjerne venstre-rekursjon på trad. måte, se neste foil.

$term \rightarrow term \text{ multop } factor \mid factor$

$term \rightarrow factor \{ \text{mulop } factor \}$

```
procedure term ;
begin
  factor ;
  while token = * do
    match (token) ;
    factor ;
  end while ;
end term ;
```



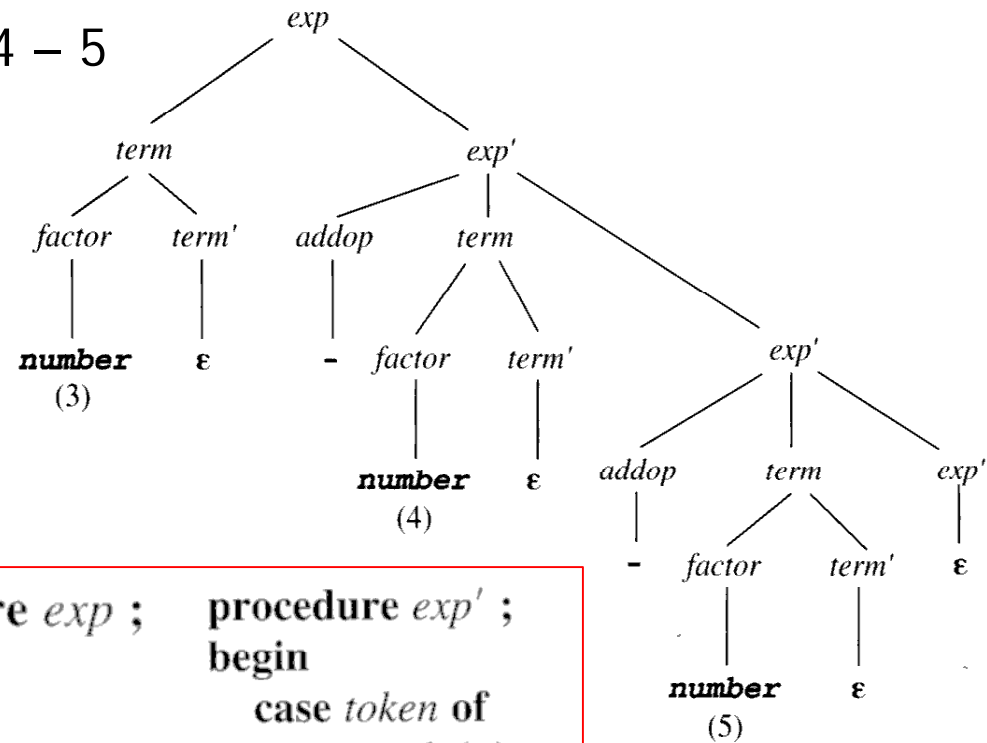


# Rec.decent etter tradisjonell fjerning av venstre-rekursjon

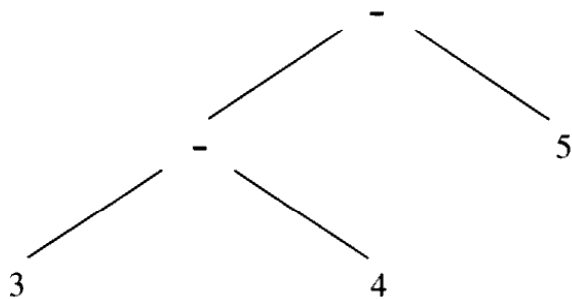
(Treet blir nå høyre assosiativt istedenfor venstre. Det må korrigeres for, se senere)

Uttrykk: 3 - 4 - 5

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



Det abstrakte syntakstreet vi egentlig ønsket å lage:



```

procedure exp ;
begin
    term ;
    exp' ;
end exp ;

procedure exp' ;
begin
    case token of
      + : match (+) ;
        term ;
        exp' ;
      - : match (-) ;
        term ;
        exp' ;
    end case ;
end exp' ;
    
```

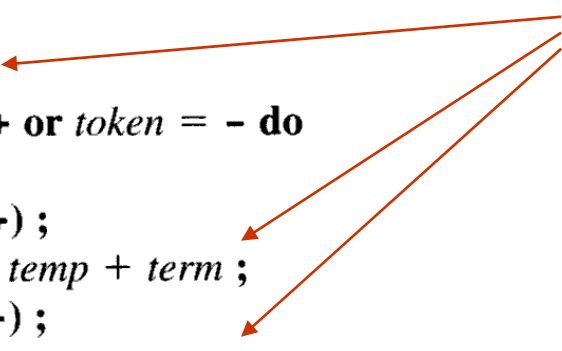


## Hvordan "lage noe" under rec.-decent parsing?

- Mål: Ønsker å bygge abstrakt syntaks-tre
- Men foreløpig (som kan være forvirrende!):
  - beregner verdien av et uttrykk (med venstre-assosiativitet)

```
function exp : integer ;  
var temp : integer ;  
begin  
  temp := term ;  
  while token = + or token = - do  
    case token of  
      + : match (+) ;  
          temp := temp + term ;  
      - : match (-) ;  
          temp := temp - term ;  
    end case ;  
  end while ;  
  return temp ;  
end exp ;
```

Kall!



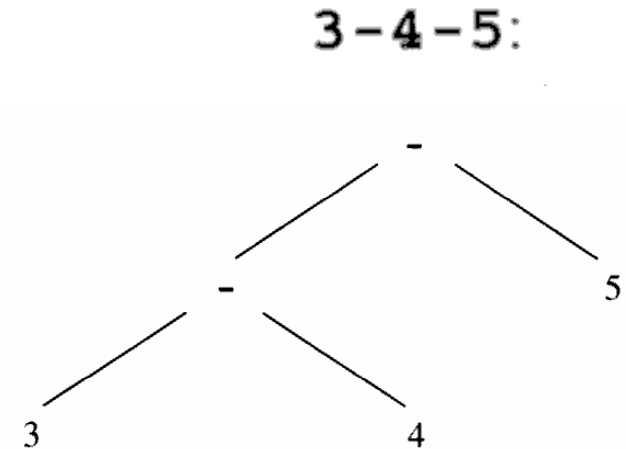
- Kan lett bygges ut til full "kalkulator"

3 + 4 + 5

# Bygging av abstrakt syntaks-tre

```
function exp : syntaxTree ;  
var temp, newtemp : syntaxTree ;  
begin  
  temp := term ; ← NB: Kall  
  while token = + or token = - do  
    case token of  
      + : match (+) ;  
        newtemp := makeOpNode(+) ;  
        leftChild(newtemp) := temp ;  
        rightChild(newtemp) := term ;  
        temp := newtemp ;  
      - : match (-) ;  
        newtemp := makeOpNode(-) ;  
        leftChild(newtemp) := temp ;  
        rightChild(newtemp) := term ;  
        temp := newtemp ;  
    end case ;  
  end while ;  
  return temp ;  
end exp ;
```

Alternativt:  
*newtemp.leftChild*



NB: Kall

Merk: Dersom det bare er én "term", så lages ingen ny node. Vi leverer den vi har fått

# Skriv prosedyre med trebygging for:

$factor \rightarrow (exp) / \underline{number}$

```
function factor: syntaxTree;  
var fact: syntaxTree;  
begin  
  case token of  
    (:  
  
    number :  
  
    else error(.....) ;  
  end case;  
  return fact;  
end factor;
```

# Prosedyre med trebygging for:

$factor \rightarrow (exp) / \underline{number}$

```
function factor: syntaxTree;  
var fact: syntaxTree;  
begin  
  case token of  
  (:  
    match ("(") ;  
    fact = exp ;  
    match (")") ;  
  number :  
    fact = makeNumberNode(number) ;  
    match (number) ;  
  else error(.....) ;  
  end case ;  
  return fact ;  
end factor ;
```

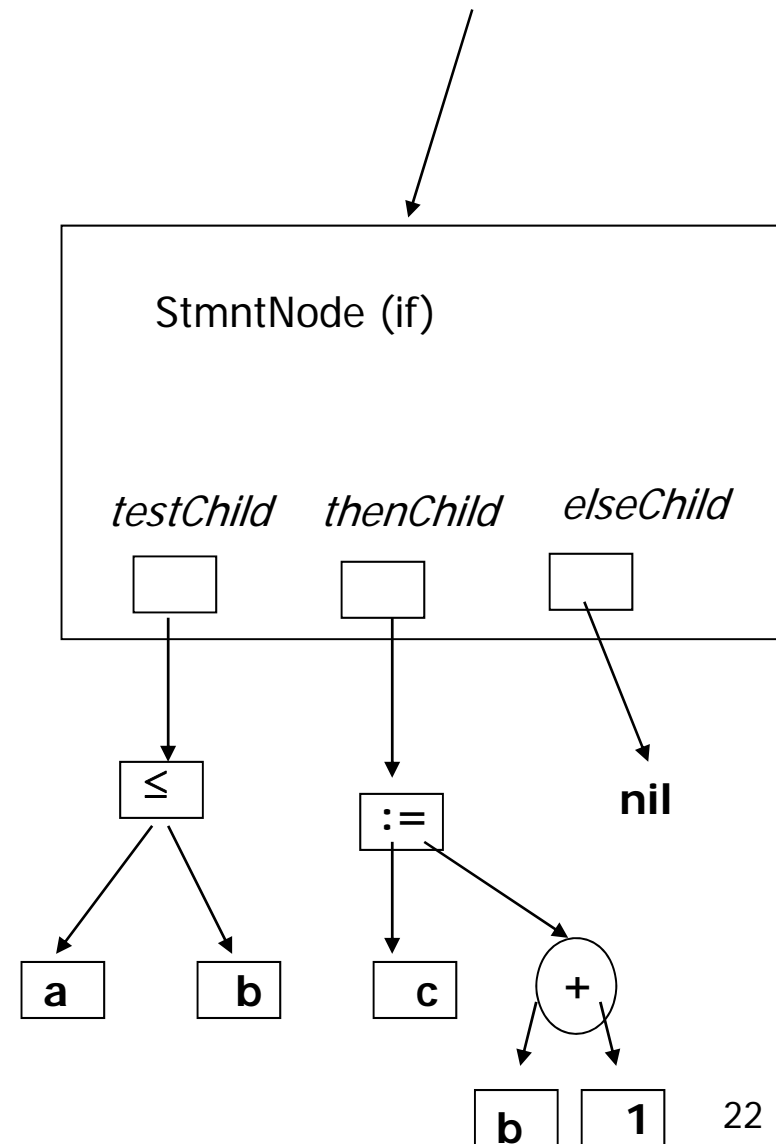
Gir "dummy-test"



# Parsering av if-setning, med tre-generering

if-stmt -> if (exp) stmt [else stmt]

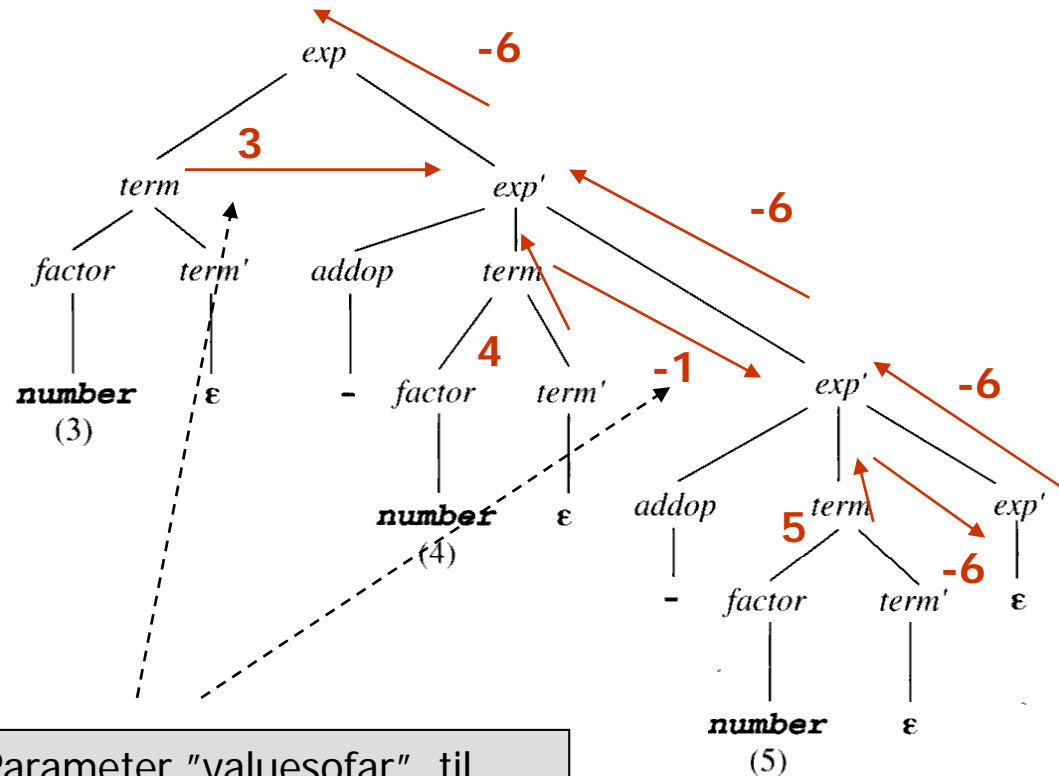
```
function ifStatement : syntaxTree ;  
var temp : syntaxTree ;  
begin  
  match (if) ;  
  match ( ) ;  
  temp := makeStmtNode(if) ;  
  testChild(temp) := exp ;  
  match ( ) ;  
  thenChild(temp) := statement ;  
  if token = else then  
    match (else) ;  
    elseChild(temp) := statement ;  
  else  
    elseChild(temp) := nil ;  
  end if ;  
end ifStatement ;
```



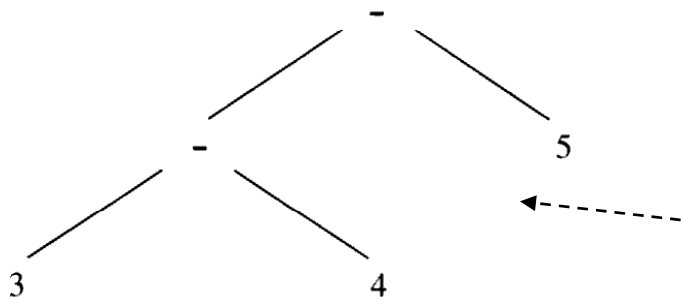
Rec.decent etter tradisjonell fjerning av venstre-rekursjon (treet er nå høyre assosiativt istedenfor venstre). Det må korrigeres for.

Lage tre eller beregne verdi : 3 - 4 - 5

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



Det abstrakte syntakstreet vi ønsker å lage:



Parameter "valuesofar" til prosedyren "exp"  
 For trebygging ville den være: "rootOfTreeSoFar"

## Prosedyrer for beregning av verdi (tre-bygging tilsvarende)

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

### Bare analyse

```
procedure exp ;  
begin  
    term ;  
    exp' ;  
end exp ;
```

```
procedure exp' ;  
begin  
    case token of  
    + : match (+) ;  
        term ;  
        exp' ;  
    - : match (-) ;  
        term ;  
        exp' ;  
    end case ;  
end exp' ;
```

### Med beregning (trebygging tilsvarende)

```
function exp : integer ;  
var temp : integer ;  
begin  
    temp := term ;  
    return exp'(temp) ;  
end exp ;
```

NB.: Parameter

```
function exp' ( valsofar : integer ) : integer ;  
begin  
    if token = + or token = - then  
        case token of  
        + : match (+) ;  
            valsofar := valsofar + term ;  
        - : match (-) ;  
            valsofar := valsofar - term ;  
        end case ;  
        return exp'(valsofar) ;  
    else return valsofar ;  
end exp' ;
```

ε -alternativet

Leverer verdien  
uendret oppover igjen.



# LL(1) – grammatikk

- LL(1) -kravet for en "ren BNF-grammatikk":

Det som kreves for at en rek. descent-parsering skal fungere direkte fra grammatikken uten omskrivninger.

- For å avgjøre om en grammatikk er "LL(1)": Sett opp tabell  $M[N,T]$  med mulige aksjoner for alle mulige situasjoner, slik:

1. If  $A \rightarrow \alpha$  is a production choice, and there is a derivation  $\alpha \Rightarrow^* a \beta$ , where  $a$  is a token, then add  $A \rightarrow \alpha$  to the table entry  $M[A, a]$ .
2. If  $A \rightarrow \alpha$  is a production choice, and there are derivations  $\alpha \Rightarrow^* \varepsilon$  and  $S \$ \Rightarrow^* \beta A a \gamma$ , where  $S$  is the start symbol and  $a$  is a token (or  $\$$ ), then add  $A \rightarrow \alpha$  to the table entry  $M[A, a]$ .

Definisjon av LL(1):

En grammatikk er LL(1) dersom  $M[A,a]$  er entydig for alle situasjoner (eller angir "error")

1. Altså, dersom

$a \in \text{First}(\alpha)$

så legg legg

$A \rightarrow \alpha$  inn i  $M[A, a]$

2. Altså, dersom:

•  $\alpha$  er utnullbar, og

•  $a \in \text{Follow}(A)$

så legg  $A \rightarrow \alpha$  inn i  $M[A, a]$

# Oppsett av LL(1) –tabell

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

- Venstre-faktorisering utført
- Er ikke vestrerekursiv

	First	Follow
statement	<b>other, if</b>	<b>\$, else</b>
if-stmt	<b>if</b>	<b>\$, else</b>
else-part	<b>else,ε</b>	<b>\$, else</b>
exp	<b>0, 1</b>	<b>)</b>

$M[N, T]$	<b>if</b>	<b>other</b>	<b>else</b>	0	1	\$
statement	statement → if-stmt	statement → <b>other</b>				
if-stmt	if-stmt → <b>if</b> ( exp ) statement else-part					
else-part			else-part → <b>else</b> statement else-part → ε			else-part → ε
exp				exp → 0	exp → 1	

**Merk:** Selv om man

- fjerner venstre-rek.
- Utfører venstre-fakt.
- er det generelt **ikke** nok til å *garantere* LL(1)-grammatikk.

**For tabellen ble ikke entydig her**

# Kan også være greit å snakke om den "utvidede startmengden", *Efirst*, til en produksjon

NB: Ikke i boka, men er pensum!

- Den "utvidede startmengden", *Efirst*, til en produksjon det som kan ligge som **førstkommende token** i input dersom  $A \rightarrow \alpha$  et riktig valg på dette stadiet under parseringen.
- Her må man også tenke på tilfellet at  $\alpha$  kan være utnullbar, og da kan også  $\text{Follow}(A)$  komme som **førstkommende token**
- Mengden kan beregnes slik:  
 $Efirst(A \rightarrow \alpha) = \text{First}(\alpha)$ , pluss, om  $\alpha$  er utnullbar,  $\text{Follow}(A)$ .
- Vi ser da at produksjonen  $A \rightarrow \alpha$  skal inn i  $M[A, a]$  hvis og bare hvis  $a \in Efirst(A \rightarrow \alpha)$
- Dermed, får vi en alternativ definisjon av LL(1):

Anta at:  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \dots \mid \alpha_n$

Da må alle:

$Efirst(A \rightarrow \alpha_1)$ ,  $Efirst(A \rightarrow \alpha_2)$ , ...,  $Efirst(A \rightarrow \alpha_n)$   
være parvis disjunkte. Merk at dette også innebærer at bare ett av alternativene er utnullbare

*På forrige side vil da både*

*$Efirst(\text{else-part} \rightarrow \text{else statmt})$*

*og*

*$Efirst(\text{else-part} \rightarrow \epsilon)$*

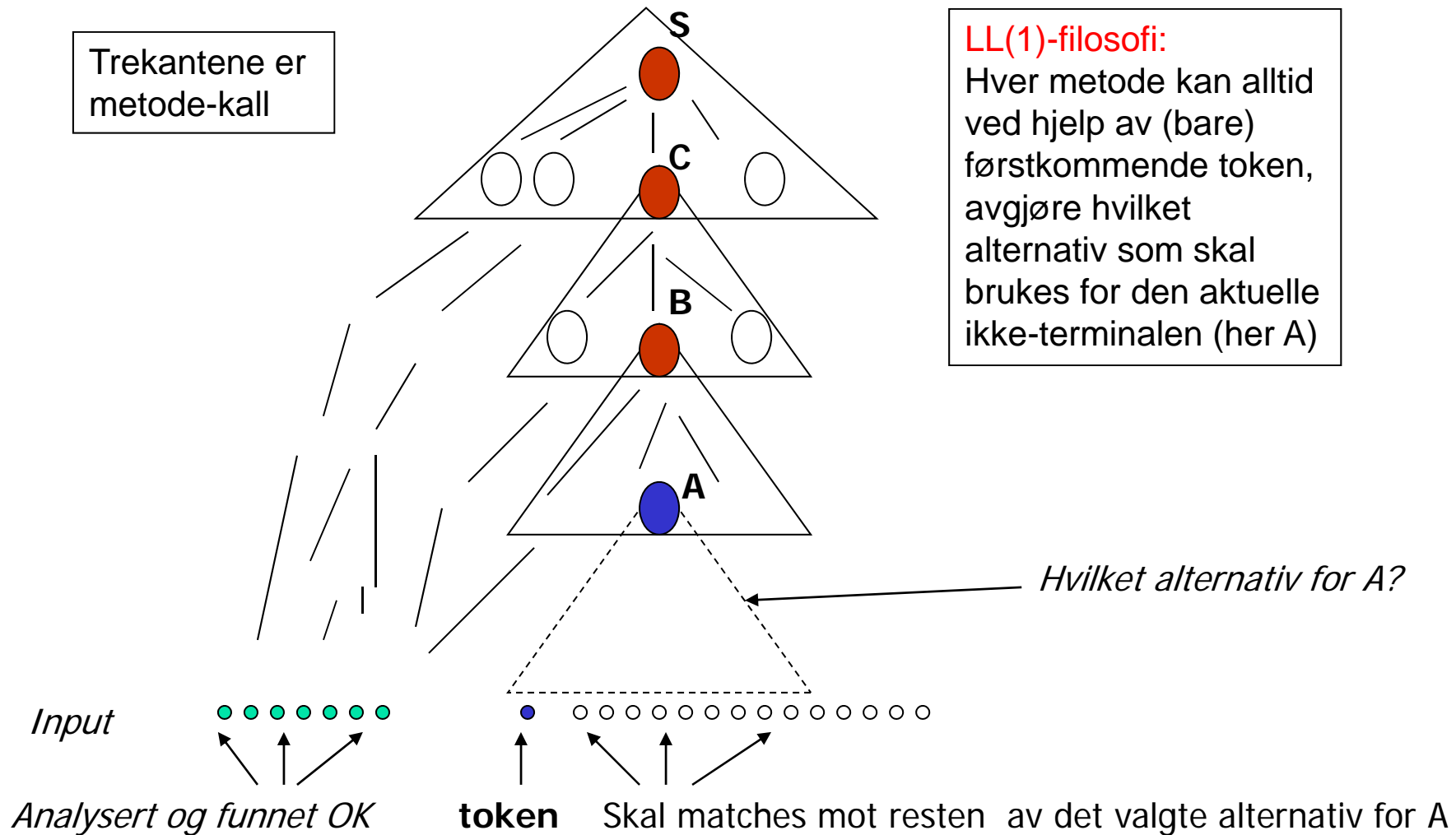
*inneholde **else***

*Dermed er grammatikken ikke LL(1)*

Tidligere foil:

# Situasjonen under rekursiv parsing

Trekantene er metode-kall



# LL(1) –tabell for uttrykks-grammatikk

Har fjernet venstre-  
rekursjon:

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

Vi får da følgende First- og  
Follow-mengder:

$$\text{First}(\text{exp}) = \{ (, \mathbf{number} \}$$
$$\text{First}(\text{exp}') = \{ +, -, \varepsilon \}$$
$$\text{First}(\text{addop}) = \{ +, - \}$$
$$\text{First}(\text{term}) = \{ (, \mathbf{number} \}$$
$$\text{First}(\text{term}') = \{ *, \varepsilon \}$$
$$\text{First}(\text{mulop}) = \{ * \}$$
$$\text{First}(\text{factor}) = \{ (, \mathbf{number} \}$$
$$\text{Follow}(\text{exp}) = \{ \$, ) \}$$
$$\text{Follow}(\text{exp}') = \{ \$, ) \}$$
$$\text{Follow}(\text{addop}) = \{ (, \mathbf{number} \}$$
$$\text{Follow}(\text{term}) = \{ \$, ), +, - \}$$
$$\text{Follow}(\text{term}') = \{ \$, ), +, - \}$$
$$\text{Follow}(\text{mulop}) = \{ (, \mathbf{number} \}$$
$$\text{Follow}(\text{factor}) = \{ \$, ), +, -, * \}$$

I

$M[N, T]$	(	<b>number</b>	)	+	-	*	\$
<i>exp</i>	<i>exp</i> → <i>term exp'</i>	<i>exp</i> → <i>term exp'</i>					
<i>exp'</i>			<i>exp'</i> → $\epsilon$	<i>exp'</i> → <i>addop</i> <i>term exp'</i>	<i>exp'</i> → <i>addop</i> <i>term exp'</i>		<i>exp'</i> → $\epsilon$
<i>addop</i>				<i>addop</i> → +	<i>addop</i> → -		
<i>term</i>	<i>term</i> → <i>factor</i> <i>term'</i>	<i>term</i> → <i>factor</i> <i>term'</i>					
<i>term'</i>			<i>term'</i> → $\epsilon$	<i>term'</i> → $\epsilon$	<i>term'</i> → $\epsilon$	<i>term'</i> → <i>mulop</i> <i>factor</i> <i>term'</i>	<i>term'</i> → $\epsilon$
<i>mulop</i>						<i>mulop</i> → *	
<i>factor</i>	<i>factor</i> → ( <i>exp</i> )	<i>factor</i> → <b>number</b>					



# Når kompilatoren oppdager feil

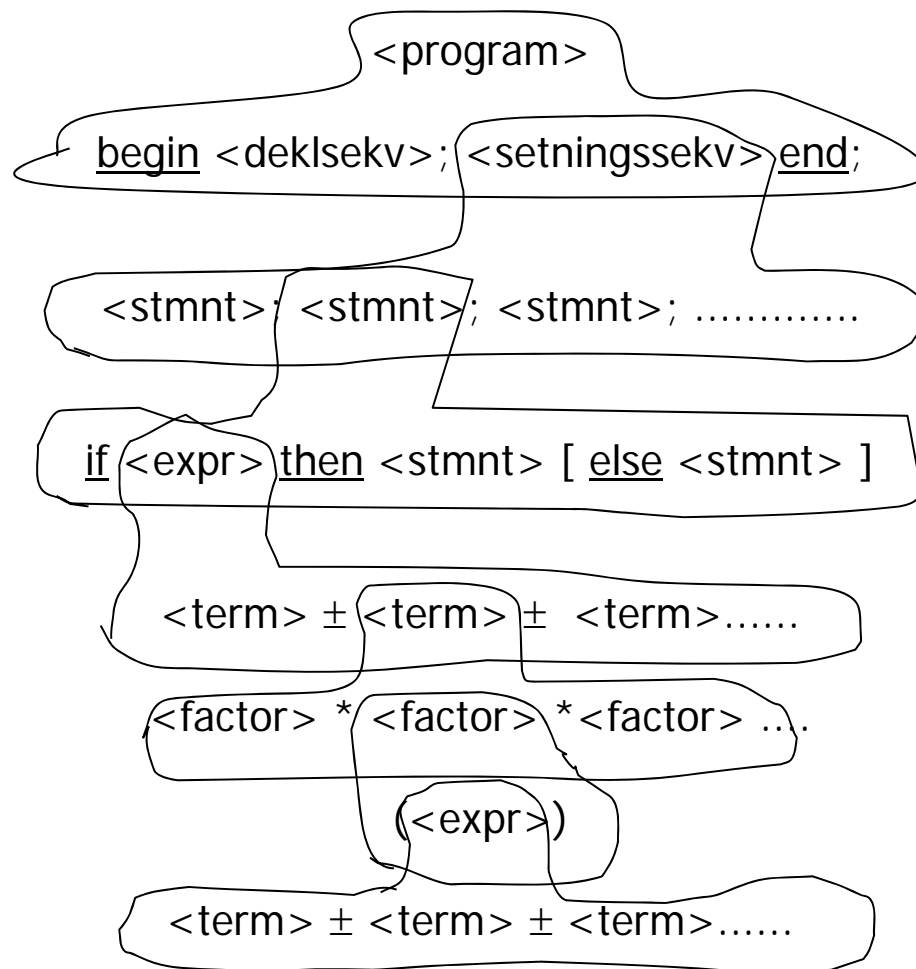
---

- Minstekrav:
  - Tester løpende at programmet er OK, og gir fornuftig feilmelding ved feil (men stopper)
- Vanlig krav ved feil ("error recovery"):
  - Gir fornuftig feilmelding.
  - "Blar forbi feilen" og fortsetter kompileringen (blar forbi så lite som mulig).
  - Vanligvis vil man slutte å lage maskinkode etter feil (Men noe "feilrettende" kompilatorer forsøker det – lite brukt)
  - Det er for *syntaksfeil* det er vanskeligst å ta opp tråden etter feil.
- Viktig:
  - Forsøke å unngå feilmeldinger som bare er følgefeil
  - Rapportere feil så tidlig som mulig, helst så snart det man har lest *ikke kan forlenges til et riktig program*
  - Man må passe på at man ikke blir gående i løkke rapportere feil *uten å lese noe fra input.*

# Behandling av Syntaksfeil

ved "recursive decent" parsering.

Metode: "Panic mode" og synkroniserings-mengde



**Synchset (stakk eller parameter):**

end

; First(stmt)

↙ navn if while for ...

then First(stmt) else

+ - First(term)

↙ ( tall navn

\* First(factor)

)

+ - ( tall navn





## Syntaksfeil ved "rec. descent" – 2

---

Ut fra skissen er det greit å finne:

- hvem som skal ta opp tråden
- "hvor" denne skal fortsette eksekveringen

Vi antar at \$ bare legges på stakken av start-symbol-metoden  
Unionen av alle på stakken kalles "synkroniseringsmengden", SM

Algoritme:

For hvert input-symbol framover, test om det er med i SM

I så fall:

- Let gjennom SM-stakken, og finn den metoden som sist ble kalt, og som kan ta opp tråden på dette input-symbolet
- Denne metoden vet selv hvor den skal fortsette, ut fra input-symbolet

Det som ikke er greit, er å programmere dette uten at den vakre strukturen ved "rec. descent" blir helt ødelagt.

# Uttrykksprosedyrer ved "error recovery"

## Filosofien her er litt annerledes (og noe uklar?)

```
procedure exp ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    term ( synchset ) ;
    while token = + or token = - do
      match ( token ) ;
      term ( synchset ) ;
    end while ;
    checkinput ( synchset, { (, number } ) ;
  end if ;
end exp ;
```

Også { +, - } ?

if token in {(,number} then ...

### Hovedfilosofi:

"checkinput" kalles to ganger: Først for å sjekke at konstruksjonen starter riktig, etterpå for å sjekke at symbolet etter konstruksjonen er lovlig.

Bruker parameter, ikke stakk  
Prosedyrene må selv ta opp tråden riktig når de får igjen kontrollen:

match(t) er som før:

- tester input mot t
- kaller eventuelt "error" (som nå returnerer!)
- kaller ikke "scanto(...)

```
procedure factor ( synchset ) ;
begin
  checkinput ( { (, number }, synchset ) ;
  if not ( token in synchset ) then
    case token of
      ( : match ( ( ) ;
        exp ( { } ) ) ;
        match ( ) ;
      number :
        match ( number ) ;
      else error ;
    end case ;
    checkinput ( synchset, { (, number } ) ; *
  end if ;
end factor ;
```

Hvorfor ikke også "synchset"?

```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset  $\cup$  { $ } ) do
    getToken ;
  end scanto ;
```

```
procedure checkinput ( firstset, followset ) ;
begin
  if not ( token in firstset ) then
    error ;
    scanto ( firstset  $\cup$  followset ) ;
  end if ;
end ;
```