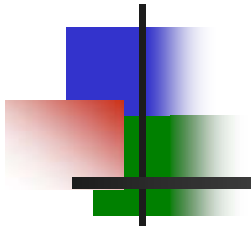
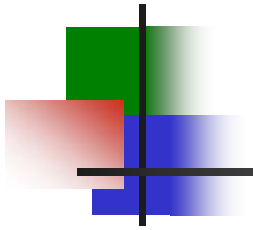


Kap. 4, del I: Top Down Parsering
Pluss oppgave til kapittel 3
INF5110 – V2008

Stein Krogdahl

Ifi, UiO

NB: Av de foilene som ble delt ut på papir på forelesningen 5/2 så utgår nr 39 – 43. Foil 44 er tatt med på starten av neste utdeling. Her er alle etter foil 38 fjernet



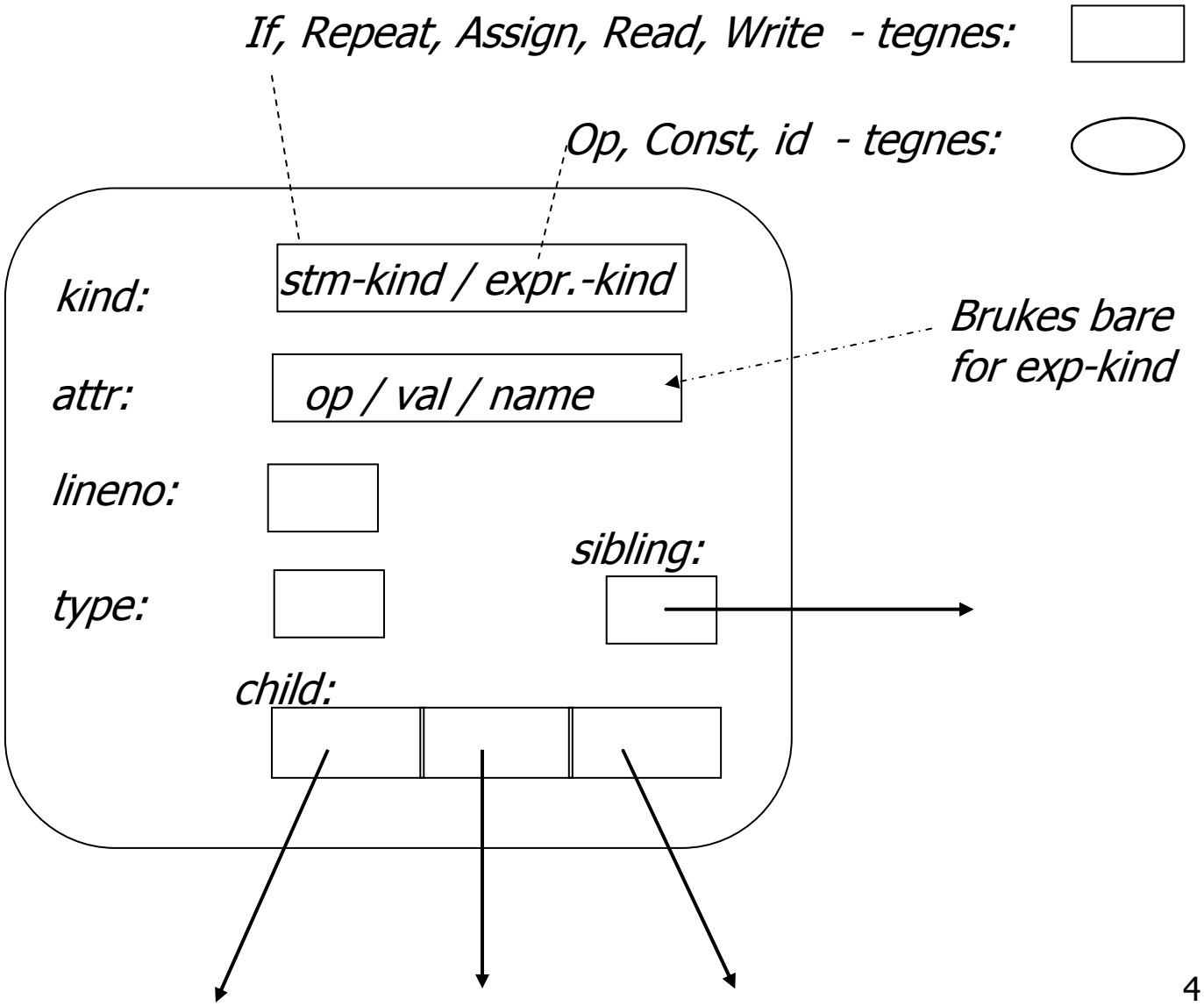
Dagens temaer

- Noen oppgaver fra sist
- First og Follow-mengder
  - Boka ser på én parseringsmetode først, uten å se på First/Follow-mengder.
  - Vi tar dette først (hører naturlig til i kap 3, Louden)
- To greie transformasjoner på grammatikker
  - Også naturlig i kap 3
  - Fjerning av venstre-rekursjon
  - Venstre-faktorisering
- Parsering ved *rekursiv nedstigning* ("recursive descent")
  - Med bygging av tre-struktur
- LL(1) grammatikker

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***

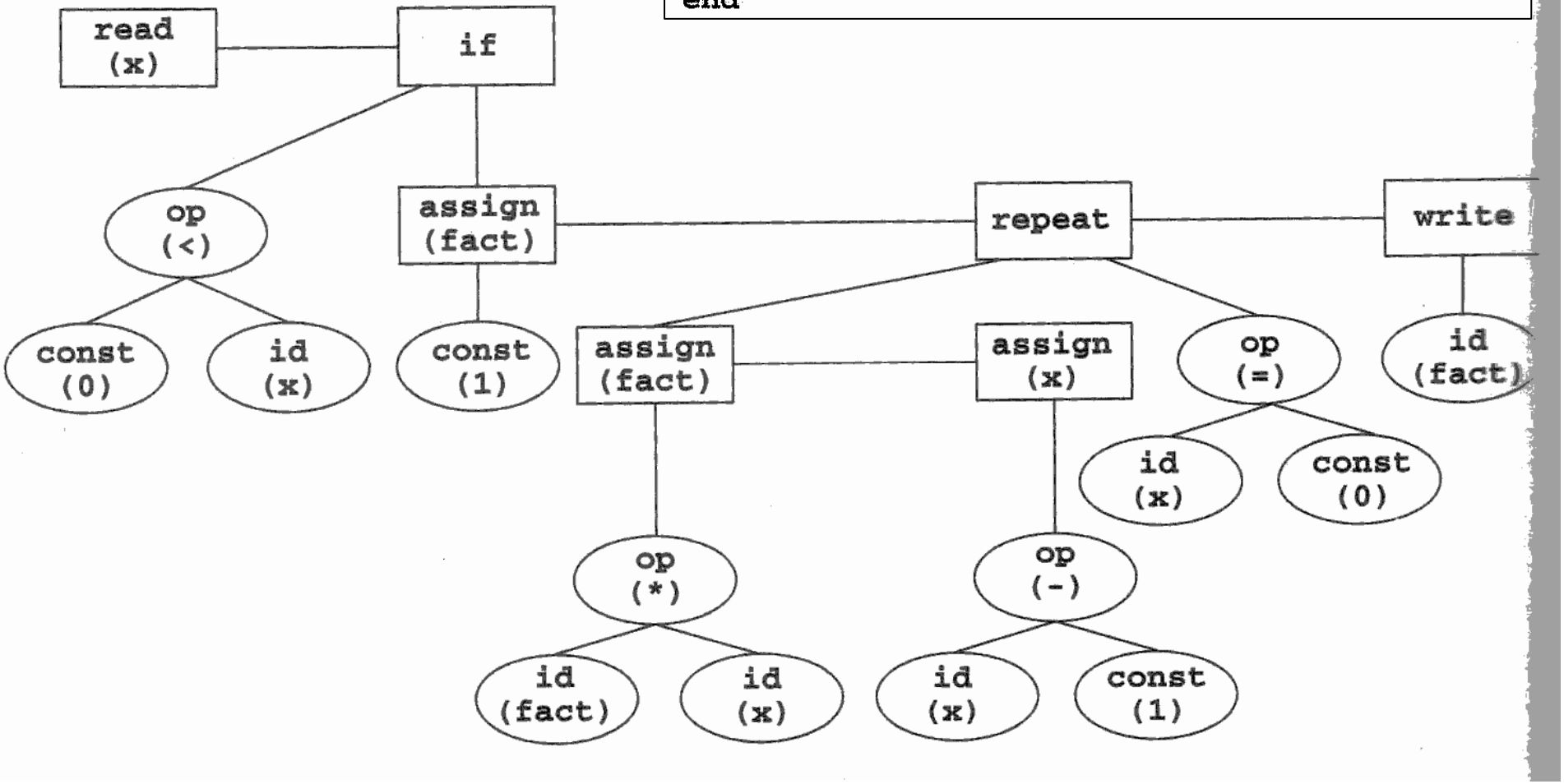
Nodestruktur i C for Tiny



Denne nodestrukturen passer enda bedre med et OO-språk med klasser /sublklasser **som implementasjons-språk.**

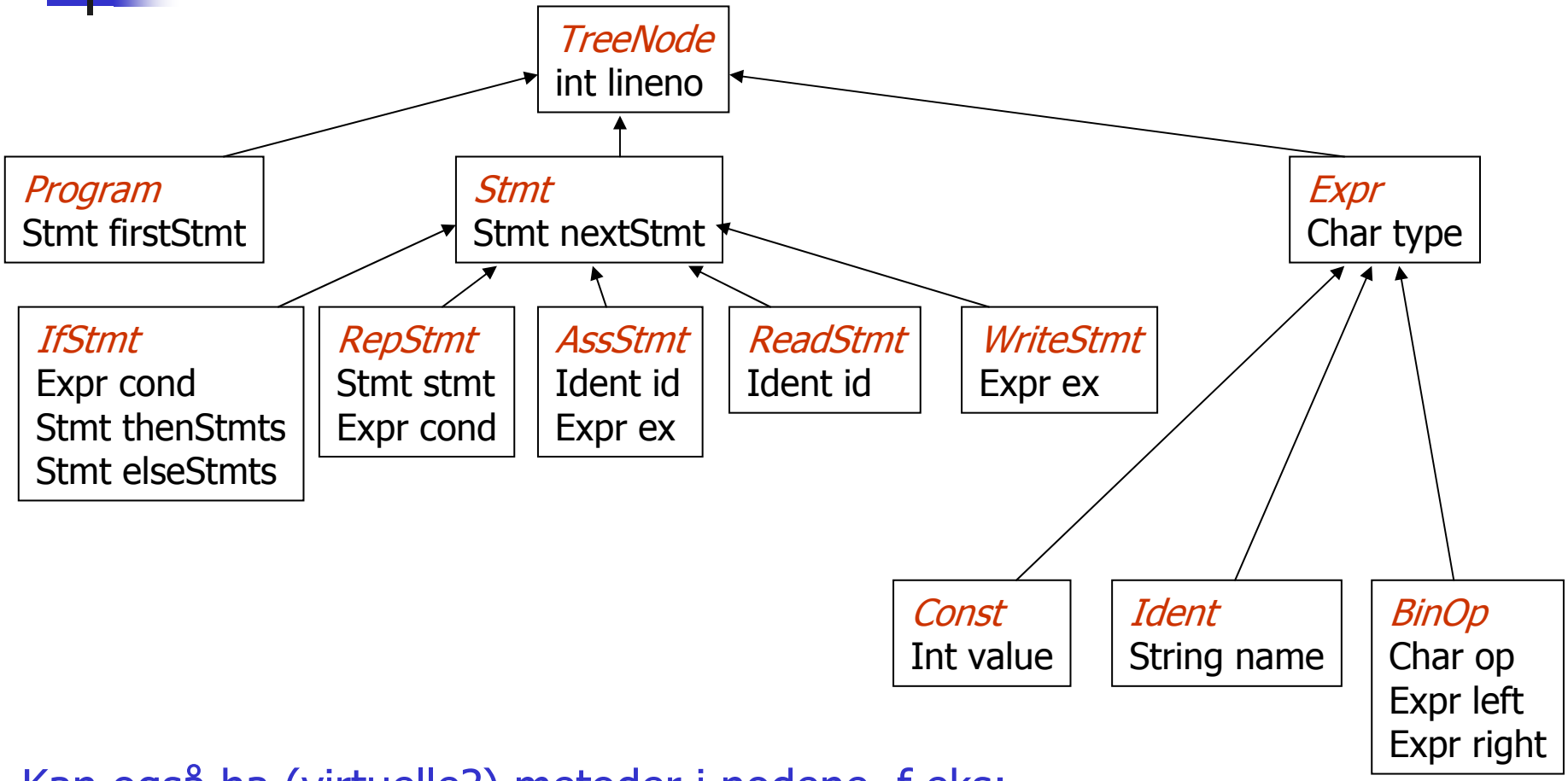
Abstrakt syntakstre for Tiny-programmet:

```
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
```



Nodeklasser til abst.synt.tre for Tiny-språket

Merk: Dette er altså en fast subklasse-struktur i kompilatoren, og må ikke forveksles med et abstrakt syntaks-tre for et eller annet program



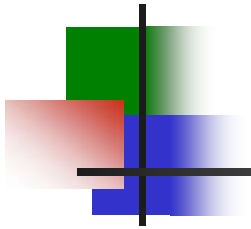
Kan også ha (virtuelle?) metoder i nodene, f.eks:

- doSemAnalyses(); Gjør semantisk analyse av noden, og av nodene under
- generateCode(); Genererer kode for noden, og for nodene under

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 assossiativitet og presedens er det for operatorene?*

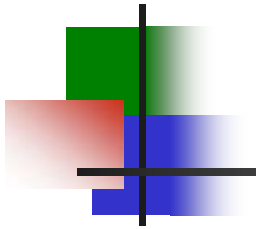


Svar på spørsmålene på forrige foil

- 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 assossiattvites-angivelse, og er nok derved entydig
- 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 - > statement ; | stmt-sequence statement ;
- Hva slags assossiativitet og presedens er det for operatorene?
  - Høyest presedens * / Venstre-assossiativ
  - Midlere presedens + - Venstre-assossiativ
  - Lavest presedens < = Ikke-assossiativitet (bare to operander)
  - Kunne altså brukt flertydig grammatikk, med disse tilleggs-reglene

First og Follow-mengder, hvorfor?



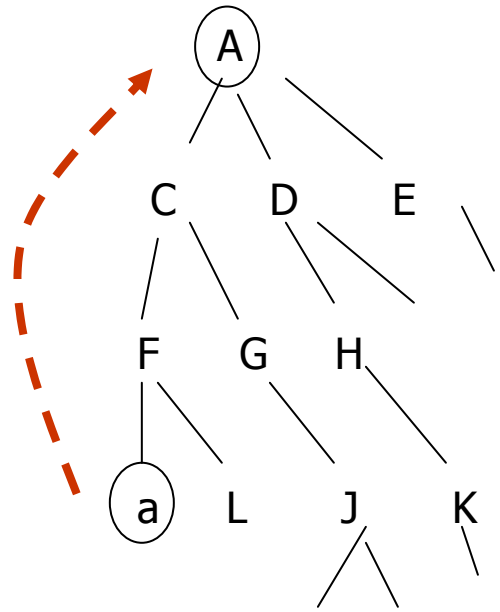
- For visse typer syntaksanalyse er det nødvendig å vite hvilke terminalsymboler som kan komme først i strenger som er avledet fra en ikketerminal A. Denne mengden kalles *First(A).*
  - F.eks. vil *First(if-setning)* i de fleste språk være bare nøkkelordet *"if"*, mens *First(tilordning)* ofte vil være mengden {*navn, "("*}.

- Tilsvarende vil vi få behov for å vite hvilke terminaler som kan følge etter en gitt ikketerminal A i *en eller annen* setningsform (="halvutviklet setning") i språket. Denne mengden kalles *Follow(A)*
  - F.eks vil *Follow(statement)* i Tiny-språket (eller snarere i den gitte *grammatikken* for Tiny-språket) være mengden {*";" , end", "else", "until"*}

- Vi skal se på generelle algoritmer for å finne First- og Follow-mengdene for alle ikke-terminaler i en gitt grammatikk
  - Det som kompliserer disse algoritmene noe er produksjoner av typen $A \rightarrow \varepsilon$, som også gjør at vi kan få $B \Rightarrow^* \varepsilon$, selv om ikke $B \rightarrow \varepsilon$ direkte. Slike ikketerminaler A eller B sies å være *utnullbare* ("nullable")

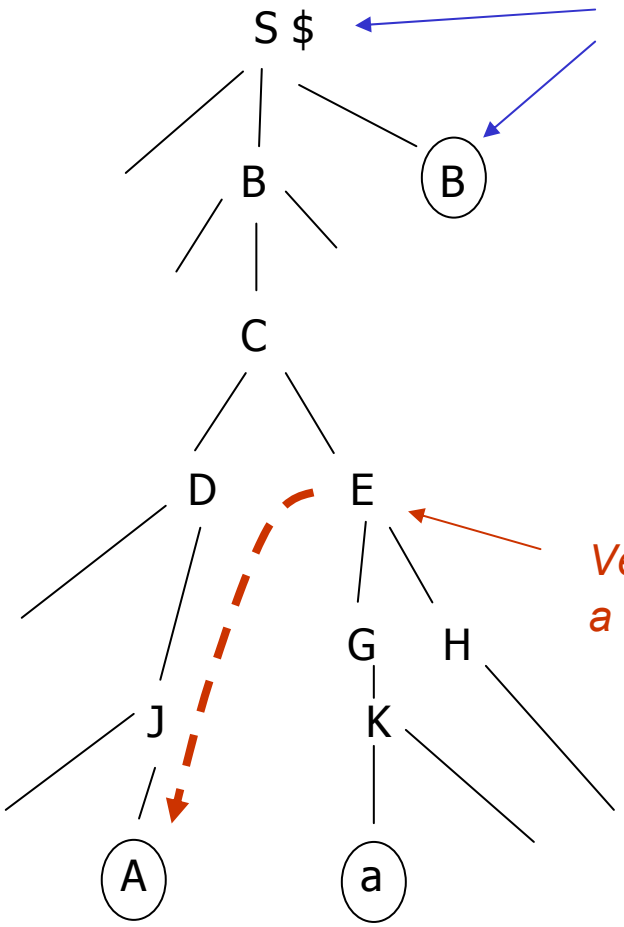
Typiske situasjoner: First- og Follow-mengder

a ∈ First(A)



Angir informasjonsflyt under algoritmene

a ∈ Follow(A)



$ ∈ Follow(B)

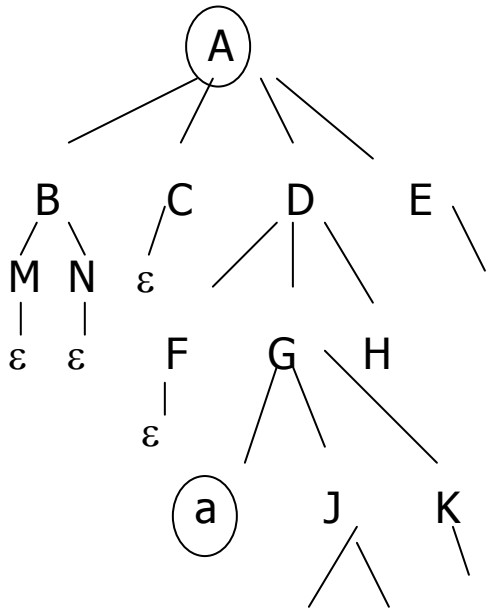
Det at B kan stå på slutten av en setnings-form markeres ved at Follow-mengden til B inneholder $.

Terminal-symbolet $ tenkes altså å stå på slutten av enhver setning.

Vet fra før at
a ∈ **First(E)**

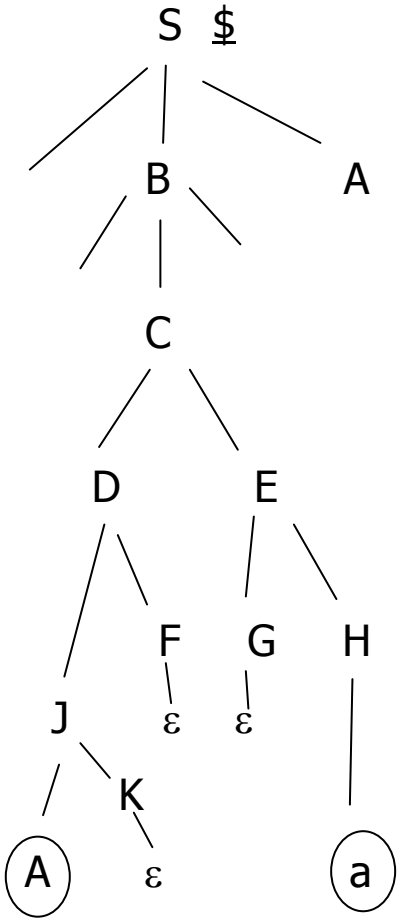
Kompliserte situasjoner: First- og Follow-mengder

a ∈ First(A)



Her er både B, M, N, C og F utnullbare. Vi markerer det ved å la deres First-mengde inneholde ε

a ∈ Follow(A)



First-mengder

terminal

Def.: { First (A) = { a | finnes avledning $A \Rightarrow^* a\,\alpha$ }
Dessuten: Om A "er utnullbar", så er $\varepsilon \in$ First(A)
Pr. def. For terminal-symboler: First(a) = { a }

Def.: "A er utnullbar" hvis og bare hvis $A \Rightarrow^* \varepsilon$

Let X be a grammar symbol (a terminal or nonterminal) or ε. Then the set **First(X)** consisting of terminals, and possibly ε, is defined as follows.

1. If X is a terminal or ε, then First(X) = $\{X\}$.
2. If X is a nonterminal, then for each production choice $X \rightarrow X_1X_2 \ldots X_n$, First($X$) contains First($X_1$) $- \{\varepsilon\}$. If also for some $i < n$, all the sets First(X_1), . . . , First(X_i) contain ε, then First(X) contains First(X_{i+1}) $- \{\varepsilon\}$. If all the sets First(X_1), . . . , First(X_n) contain ε, then First(X) also contains ε.

Now define **First(α)** for any string $\alpha = X_1X_2 \ldots X_n$ (a string of terminals and nonterminals), as follows. First(α) contains First(X_1) $- \{\varepsilon\}$. For each $i = 2, \ldots, n$, if First(X_k) contains ε for all $k = 1, \ldots, i - 1$, then First(α) contains First(X_i) $- \{\varepsilon\}$. Finally, if for all $i = 1, \ldots, n$, First(X_i) contains ε, then First(α) contains ε.

← Tar vi som *algoritme* for å finne *First( )*

← Vi snakker altså også om *First* av en hel streng α av terminaler og ikke-terminaler

```
for all nonterminals A do First(A) := {};
while there are changes to any First(A) do
  for each production choice A → X1X2 . . . Xn do
    k := 1 ; Continue := true ;
    while Continue = true and k <= n do
      add First(Xk)−{ε} to First(A);
      if ε is not in First(Xk) then Continue := false ;
      k := k + 1 ;
    if Continue = true then add ε to First(A) ;
```

Algoritme:

- Gjør steg 1.
- Gjenta steg 2 til alle Firstmengdene har stabilisert seg.

Eks. 4.9 Beregning av First-mengde

(1) *exp* → *exp addop term*
(2) *exp* → *term*
(3) *addop* → **+**
(4) *addop* → **-**
(5) *term* → *term mulop factor*
(6) *term* → *factor*
(7) *mulop* → *****
(8) *factor* → **(** *exp* **)**
(9) *factor* → **number**

Grammar rule	Pass 1	Pass 2	Pass 3
exp → *exp addop term*			
exp → *term*			First(*exp*) = { **(**, **number** }
addop → **+**	First(*addop*) = {**+**}		
addop → **-**	First(*addop*) = {**+**, **-**}		
term → *term mulop factor*			
term → *factor*		First(*term*) = { **(**, **number** }	
mulop → *****	First(*mulop*) = {*****}		
factor → **(** *exp* **)**	First(*factor*) = { **(** }		
factor → **number**	First(*factor*) = { **(**, **number** }		

Kan enklere fylle ut en tabell:

	First
exp	
addop	
term	
multop	
factor	

NB: Kan ta reglene i vilkårlig rekkefølge, frem og tilbake

Beregning av Follow-mengder

β γ virkårlige strenger

Def: Follow (A) = { a | finnes avledning S $\Rightarrow^*$ β A a γ }
For stratsymbolet S er $ med i followmenden

Dvs.: Det finnes en avledning fra setningsformen "S $" til en setningsform hvor A kommer rett før a (a er en terminal)

Given a nonterminal A, the set **Follow(A)**, consisting of terminals, and possibly \$, is defined as follows.

1. If A is the start symbol, then \$ is in Follow(A).
2. If there is a production $B \rightarrow \alpha\ A\ \gamma$, then First($\gamma$) $-$ $\{\varepsilon\}$ is in Follow(A).
3. If there is a production $B \rightarrow \alpha\ A\ \gamma$ such that ε is in First (γ), then Follow(A) contains Follow(B).

γ er utnullbar

```
Follow(start-symbol) := {$} ;
for all nonterminals A ≠ start-symbol do Follow(A) := {} ;
while there are changes to any Follow sets do
  for each production A → X1X2 . . . Xn do
    for each Xi that is a nonterminal do
      add First(Xi+1Xi+2 . . . Xn) − {ε} to Follow(Xi)
      (* Note: if i=n, then Xi+1Xi+2 . . . Xn = ε *)
      if ε is in First(Xi+1Xi+2 . . . Xn) then
        add Follow(A) to Follow(Xi)
```

Algoritme:

- **Gjør steg 1.**
- **Gjenta steg 2 og 3 til alle Follow-mengdene har stabilisert seg.**

Beregning av Follow-mengder: 4.12

(1) *exp* → *exp addop term*
(2) *exp* → *term*
(3) *addop* → **+**
(4) *addop* → **-**
(5) *term* → *term mulop factor*
(6) *term* → *factor*
(7) *mulop* → *****
(8) *factor* → **(*exp*)**
(9) *factor* → **number**

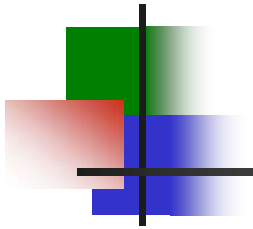
First(*exp*) = { **(**, **number** }
First(*term*) = { **(**, **number** }
First(*factor*) = { **(**, **number** }
First(*addop*) = {**+**, **-**}
First(*mulop*) = {*****}

Follow(*exp*) = {$, **+**, **-**, **)** }
Follow(*addop*) = { **(**, **number** }
Follow(*term*) = {$, **+**, **-**, *****, **)** }
Follow(*mulop*) = { **(**, **number** }
Follow(*factor*) = {$, **+**, **-**, *****, **)** }

Grammar rule	Pass 1	Pass 2
exp → *exp addop term*	Follow(*exp*) = {$, **+**, **-**} Follow(*addop*) = { **(**, **number**} Follow(*term*) = {$, **+**, **-**}	Follow(*term*) = {$, **+**, **-**, *****, **)** }
exp → *term*		
term → *term mulop factor*	Follow(*term*) = {$, **+**, **-**, *****} Follow(*mulop*) = { **(**, **number**} Follow(*factor*) = {$, **+**, **-**, *****}	Follow(*factor*) = {$, **+**, **-**, *****, **)** }
term → *factor*		
factor → **(** *exp* **)**	Follow(*exp*) = {$, **+**, **-**, **)** }	

Kan bare fylle ut en tabell:

	Follow
exp	
addop	
term	
multop	
factor	



Et par greie transformasjoner

- For noen typer syntaksanalyse er det visse krav til grammatikken for at metoden skal fungere. To vanlige krav er:
  - Grammatikken må ikke ha venstrerekursjon
    - Altså, det må ikke finnes produksjoner av typen: $A \rightarrow A\ \alpha \mid \ldots$
    - F.eks. går da ikke: expr $\rightarrow$ expr + term | term
  - Grammatikken må ikke ha noen B (ikke-terminal), slik at to alternativer for B starter på samme måte.
    - Altså, det må ikke finnes slike: $B \rightarrow \alpha\ \beta \mid \ldots \mid \alpha\ \gamma \mid \ldots \qquad \alpha \neq \varepsilon$
    - F.eks. går da ikke:

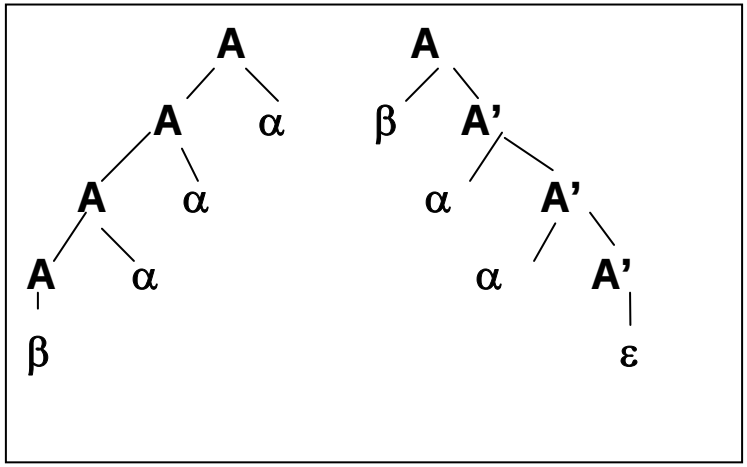
      if-setn $\rightarrow$ **if** (bet) setn-sekv **end** | **if** (bet) setn-sekv **else** setn-sekv **end**

Fjerning av venstre-rekursjon - case 1

Kan skrives på EBNF

$A \rightarrow A\alpha \mid \beta$

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



$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

NB: Litt dumt, siden den opprinnelige venstreassossiative strukturen forsvinner. Må ev. korrigeres for.

Eksempel:

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

This is of the form $A \rightarrow A\ \alpha \mid \beta$, with $A = exp$, $\alpha = addop\ term$, and $\beta = term$. Rewriting this rule to remove left recursion, we obtain

$$exp \rightarrow term\ exp'$$
$$exp' \rightarrow addop\ term\ exp' \mid \varepsilon$$

Fjerning av venstre-rekursjon – case 2

Case 2:

$$A \rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \ldots \mid A\,\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$$

Kan skrives: A -> (β1 | β2 | … | βm) (α1 | α2 | … | αn)*

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon$$

Case 3 (indirekte venstre-rekursjon)

A → Ba | Aa | c
B → Bb | Ab | d

- **Her er det feil i boka i algoritmen s.159**
- **Ikke med som pensum**
- **Egner seg bare til maskinell behandling**

$$A \rightarrow B\,a\,A' \mid c\,A'$$
$$A' \rightarrow a\,A' \mid \varepsilon$$
$$B \rightarrow B\,b \mid A\,b \mid d$$

$$A \rightarrow B\,a\,A' \mid c\,A'$$
$$A' \rightarrow a\,A' \mid \varepsilon$$
$$B \rightarrow B\,b \mid B\,a\,A'\,b \mid c\,A'\,b \mid d$$

$$A \rightarrow B\,a\,A' \mid c\,A'$$
$$A' \rightarrow a\,A' \mid \varepsilon$$
$$B \rightarrow c\,A'\,b\,B' \mid d\,B'$$
$$B' \rightarrow b\,B' \mid a\,A'\,b\,B' \mid \varepsilon$$

Eks. side 160 – Fjerning av venstre-rekursjon

Den tradisjonelle entydige grammatikken

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

Med fjernet venstre-rekursjon (beholder entydigheten, men ikke assossiativiteten):

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

Venstrefaktorisering, problemet "B → α β | α γ |..."

stmt-sequence → *stmt* **;** *stmt-sequence* | *stmt*
stmt → **s**

Blir til:

stmt-sequence → *stmt stmt-seq'*
stmt-seq' → **;** *stmt-sequence* | ε

---

f-setn → **if** (*bet*) *setn-sekv* **end** | **if** (*bet*) *setn-sekv* **else** *setn-sekv* **end**

Blir til:

i*f-setn* → **if** (*bet*) s*etn-sekv* *else-eller-end*
else-eller-end → **else** setn-sekv **end** | **end**

---

if-setn → **if** (*bet*) *setn* **end** | **if** (*bet*) *setn* **else** *setn*

Blir til:

i*f-setn* → **if** (*bet*) s*etn* *else-eller-tom*
else-eller-tom → **else** setn | **ε**

Avansert venstre-faktorisering

Et litt mer komplisert tilfelle:

$A \rightarrow abc\ B \mid abC \mid aE$

Etter steg 1: $A \rightarrow ab\ A' \mid aE$

$A' \rightarrow c\ B \mid C$

Etter steg 2 (og ferdig): $A \rightarrow a\ A''$

$A'' \rightarrow b\ A' \mid E$

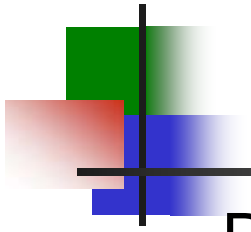
$A' \rightarrow c\ B \mid C$

while *there are changes to the grammar* **do**
  for *each nonterminal A* **do**
    let α *be a prefix of maximal length that is shared by two or more production choices for A*
    if $\alpha \neq \varepsilon$ **then**
      let $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ *be all the production choices for A and suppose that* $\alpha_1, \ldots, \alpha_k$ *share* α, *so that* $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_k \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$, *the* β_j*'s share no common prefix, and the* $\alpha_{k+1}, \ldots, \alpha_n$ *do not share* α
      replace the rule $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ *by the rules*
        $A \rightarrow \alpha A' \mid \alpha_{k+1} \mid \ldots \mid \alpha_n$
        $A' \rightarrow \beta_1 \mid \ldots \mid \beta_k$

} Gjenta så lenge det er muligheter

} Velg lengst mulig prefiks

Gjør som over. Pass på å få med alle alternativer med dette prefikset



Litt om stoffet i kap. 4:

Dette bør leses om igjen etter kapittelet:

- *First* og *Follow*-mengder
  - Boka tar det et stykke uti, vi tok det først
- Begrepet *LL(1)-parsering* (kommer)
  - Boka bruker dette begrepet mest for den metoden (kap 4.2 s. 152) der man bruker en eksplisitt stakk (isteden for rekursive metoder, som i ”recursive decent”-framgangsmåten).

    7/2- 2008: Denne metoden er nå tatt ut av pensum!
  - Det vanlige er å bruke betegnelsen LL(1)-parsering også når rec.decent metoden brukes slavisk på en ren BNF-grammatikk
  - Kravet til en LL(1)-grammatikk kommer like tydelig fram ut fra begge disse metodene. Vi skal tenke like mye rec.decent som eksplisitt stakk. (boka gjør bare det siste).
  - Ofte brukes betegnelsen LL(1)-parsering også om rec.decent-metoden brukt ut fra syntaksdiagrammer eller EBNF, men da er det uklart hva LL(1)-kravet til en grammatikk er.

Parseringsmetode: ”Rekursiv nedstigning ”

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

Hoved-idé:

- Skriv en funksjon /prosedyre/metode for hver ikke-terminal
- La denne lese/sjekke høyreside-alternativene

Inneværende token: `if` ← *Global variabel*

”Typisk” rec.decent prosedyre for det enkle tilfellet.

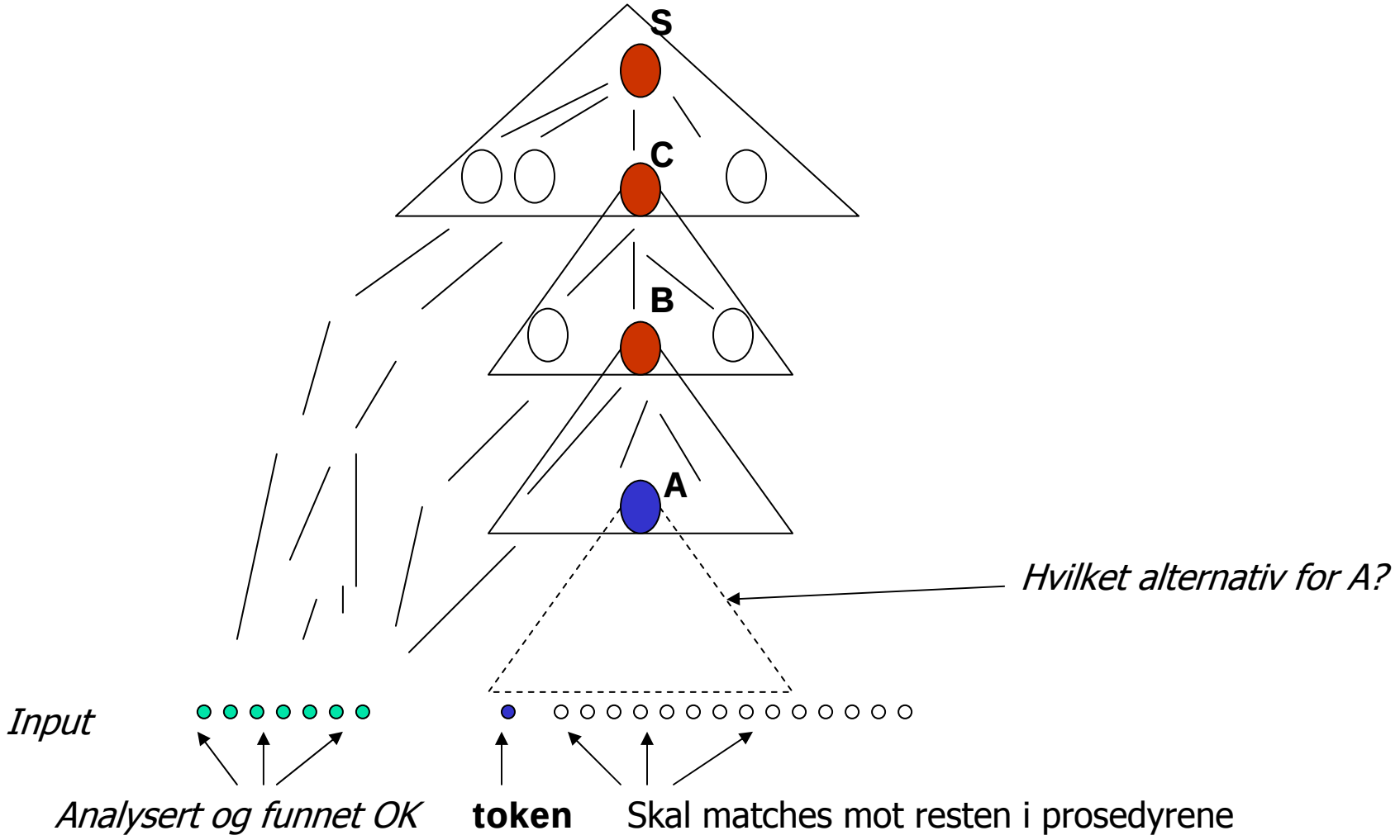
```
procedure factor ;
begin
  case token of
  (:    match( ( ) ;
        exp ;
        match( ) ) ;
  number :
        match(number) ;
  else error ;
  end case ;
end factor ;
```

Sjekker at angitt terminal kommer, og ”leser til neste”. Brukes ofte bare for å lese (sjekken må slå til).

```
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)



Litt mer kompliserte tilfeller kan løses med EBNF

Opprinnelig

if-stmt → **if** **(** *exp* **)** *statement* [**else** *statement*]

Skrives ut som:

if-stmt → **if** **(** *exp* **)** *statement*
| **if** **(** *exp* **)** *statement* **else** *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":

ifStmt → **if** (exp) stmt elsePart
elsePart → ε | **else** stmt

Venstre-rekursjon gir problemer

Gir uendelig mange rekursive kall

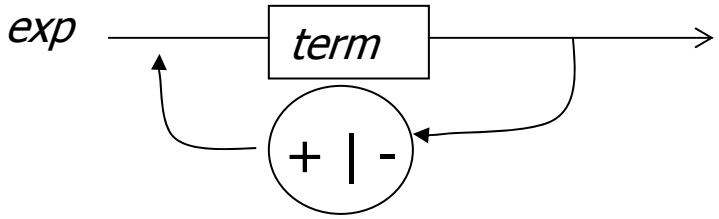
$exp \rightarrow exp\ addop\ term \mid term$

NB: Kan også fjerne venstre-rekursjon på trad måte, se senere foiler.

Bruker EBNF:

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

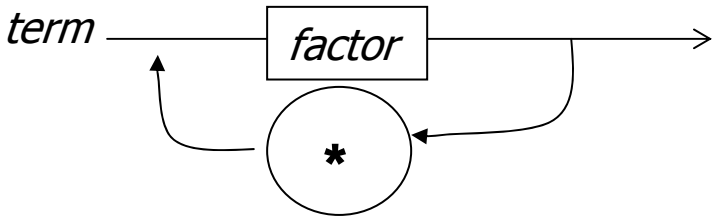
```
procedure exp ;
begin
   term ;
   while token = + or token = - do
      match (token) ;
      term ;
   end while ;
end exp ;
```



$term \rightarrow term\ multop\ factor \mid factor$

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

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



Hvordan "lage noe" under rec.-decent parsering?

- 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 ;
```

- Kan lett bygges ut til full "kalkulator" 3 + 4 + 5

Bygging av abstrakt syntaks-tre

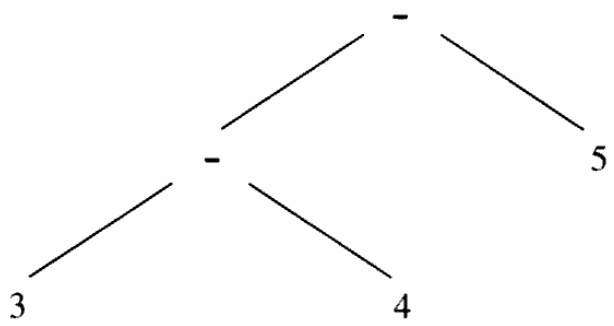
```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
   temp := term ;
   while token = + or token = - do
      case token of
       + : match (+) ;
           newtemp := makeOpNode(+) ;
           leftChild(newtemp) := temp ;
           rightChild(newtemp) := term ;
           temp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

Kall

Alternativt:
newtemp.leftChild

Kall

3-4-5:



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

Skriv prosedyre med trebygging for:

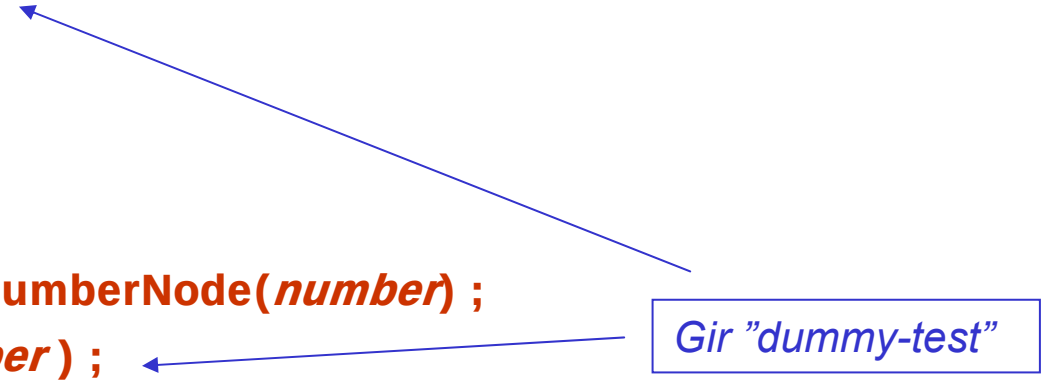
factor → *(exp)* | *number*

```
proc factor : syntaxTree;
var fact: syntaxTree;
begin
    case token of
    ( :




    number :



    else error(.....) ;
    end case
    return fact;
end factor;
```

Prosedyre med trebygging for:

factor → *(exp)* | *number*

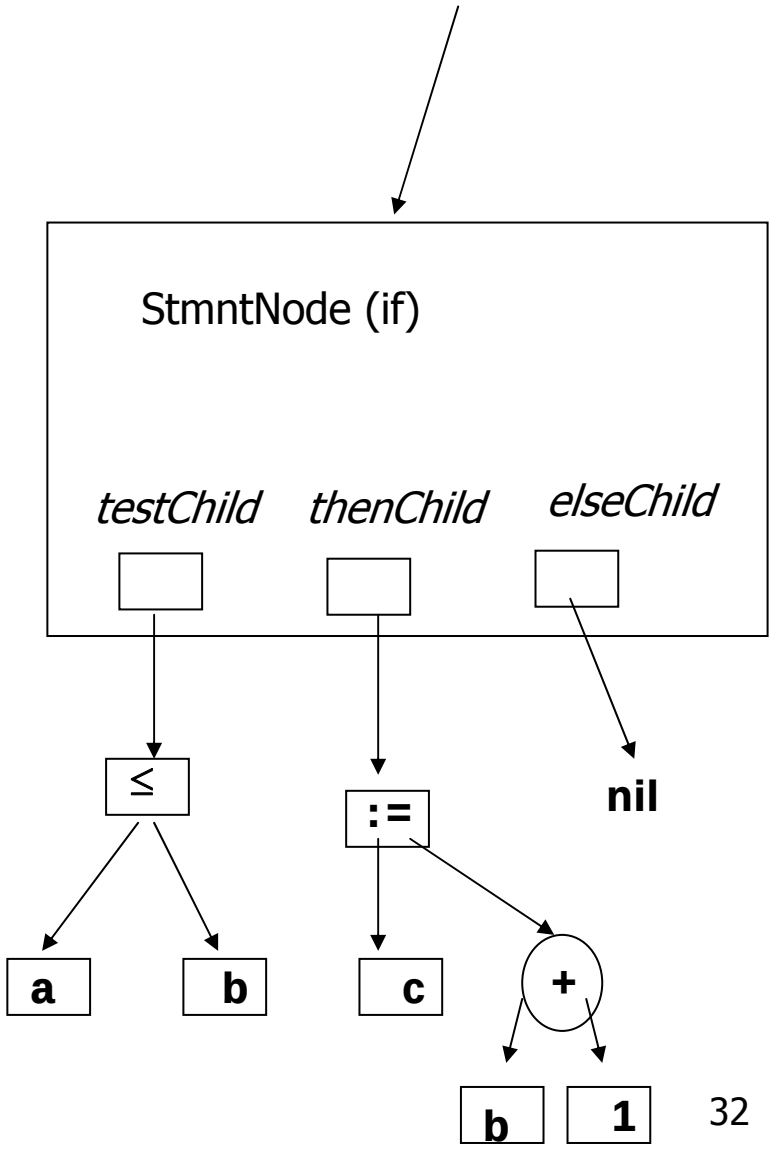
```
proc factor : syntaxTree;
var fact: syntaxTree;
begin
    case token of
    ( :
        match ("(") ;
        fact = exp ;
        match (")") ;
    number :
        fact = makeNumberNode(number) ;
        match (number ) ;
    else error(.....) ;
    end case
    return fact;
end factor;
```



Parsering av if-setning, med tre-generering

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

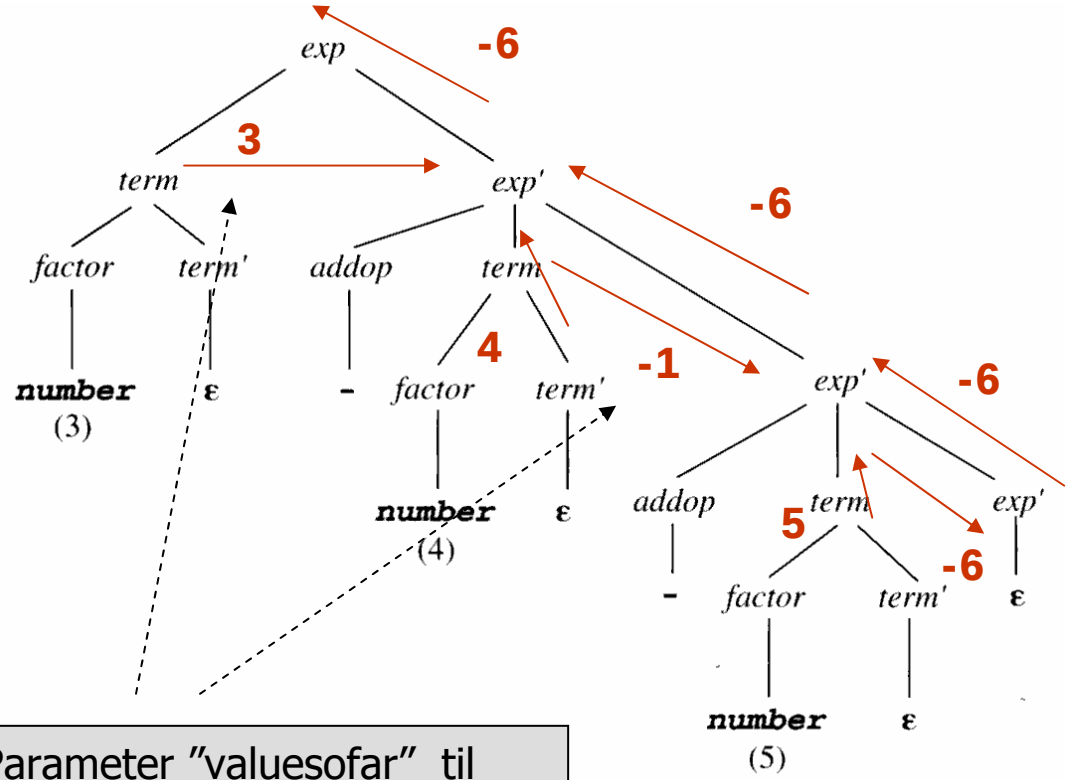
```
function ifStatement : syntaxTree ;
var temp : syntaxTree ;
begin
   match (if) ;
   match ( ( ) ;
   temp := makeStmtNode(if) ;
   testChild(temp) := exp ;
   match ( ) ) ;
   thenChild(temp) := statement ;
   if token = else then
      match (else) ;
      elseChild(temp) := statement ;
   else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



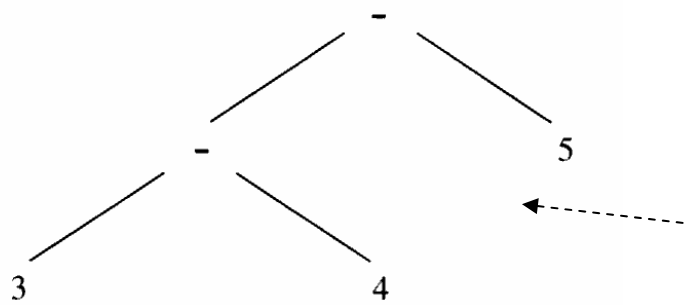
Rec.decent etter tradisjonell fjerning av venstre-rekursjon (treet er nå høyre assosiativt istedenfor venstre). Det må korrigeres for.

Lage tre eller beregne verdi : 3 – 4 – 5

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

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

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

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid .... \mid \alpha_n$

- LL(1) -kravet for en ”ren BNF-grammatikk”. Det som kreves for at en Rek. descent-parsering skal fungere direkte fra grammatikken.
- 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 $M[A, a]$.

1. Altså,
a $\in$ First(α)

2. Altså, dersom:

- α er utnullbar, og
- a $\in$ Follow(A)

Definisjon:
En grammatikk er LL(1) dersom M[N,T] er entydig for alle situasjoner (eller angir ”error”)

Oppsett av LL(1) –tabell

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

- Venstre-faktorisering utført
- Er ikke vestrerekursiv

	First	Follow
statement	**other, if**	**$, else**
if-stmt	**if**	**$, else**
else-part	**else,ε**	**$, else**
exp	**0, 1**	**)**

M[N, T]	**if**	**other**	**else**	0	1	$
statement	*statement* → *if-stmt*	*statement* → **other**				
if-stmt	*if-stmt* → **if (** *exp* **)** *statement else-part*					
else-part			*else-part* → **else** *statement* *else-part* → ε			*else-part* → ε
exp				*exp* → **0**	*exp* → **1**	

Merk:

- fjerne venstre-rek.
- Utføre venstre-fakt.
- er generelt **ikke** nok til å garantere LL(1)-grammatikk.

Fordi tabellen ikke ble entydig

LL(1) –tabell for uttrykks-grammatikk

Har fjernet venstre-rekursjon:

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

Vi får da følgende First- og Follow-mengder:

First(exp) = { **(**, **number**}	Follow(exp) = {$, **)**}
First(exp') = {**+**, **-**, ε}	Follow(exp') = {$, **)**}
First($addop$) = {**+**, **-**}	Follow($addop$) = { **(**, **number**}
First($term$) = { **(**, **number**}	Follow($term$) = {$, **)**, **+**, **-**}
First($term'$) = {*****, ε}	Follow($term'$) = {$, **)**, **+**, **-**}
First($mulop$) = {*****}	Follow($mulop$) = { **(**, **number**}
First($factor$) = { **(**, **number**}	Follow($factor$) = {$, **)**, **+**, **-**, *****}

M[N, T]	(	**number**	)	+	-	*	$
exp	*exp* → *term exp′*	*exp* → *term exp′*					
exp′			*exp′* → ε	*exp′* → *addop term exp′*	*exp′* → *addop term exp′*		*exp′* → ε
addop				*addop* → +	*addop* → -		
term	*term* → *factor term′*	*term* → *factor term′*					
term′			*term′* → ε	*term′* → ε	*term′* → ε	*term′* → *mulop factor term′*	*term′* → ε
mulop						*mulop* → *	
factor	*factor* → (*exp*)	*factor* → **number**					