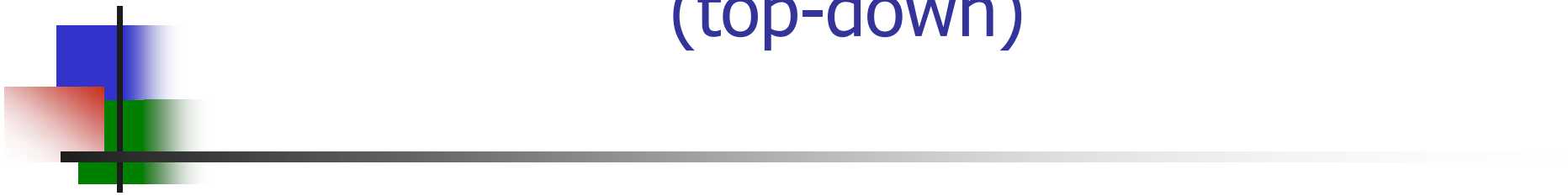
INF5110 – V2012
Kapittel 4: Parsering ovenfra-ned (top-down)



Tirsdag 7. februar

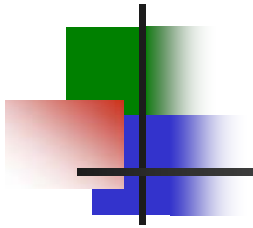
Stein Krogdahl, Ifi, UiO

Oppgaver som gjennomgås i morgen, onsdag:

-Spørsmålene på de to siste foilene fra onsdag 1/2 (Bl.a. lag et subklasse-hierarki som ville passe for nodene i et abstrakt syntakstre for Tiny)

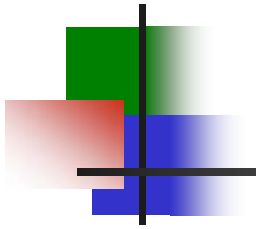
-Formuler en entydig uttrykks-grammatikk som også har (høyreassosiativ) eksponering

- Beskriv en algoritme som finner alle de ikke-terminaler som kan produsere den tomme streng (uten også å finne First-mengden). Vi kaller dem ”utnullbare”.



Dagens temaer

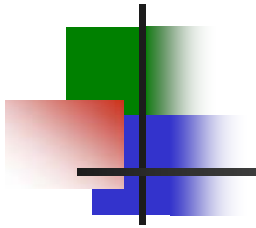
- Noe avsluttende fra sist (se lysark fra da)
- First og Follow-mengder
  - Vi tar det i en litt annen rekkefølge enn i boka
  - Vi starter i dag med First-mengder
  - Follow-mengder kommer kanskje ikke i dag
- Begynnende om parsering ovenfra-ned (top-down)
  - Recursive descent-metoden
- To greie transformasjoner på grammatikker
  - Fjerning av venstre-rekursjon
  - Venstre-faktorisering



Litt om kapittel 4, og ”LL(1)-grammatikker”

Det under bør leses om igjen etter hele kapittel 4!

- *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* ser i dette kapittelet mest på ”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



First og Follow-mengder, hvorfor?

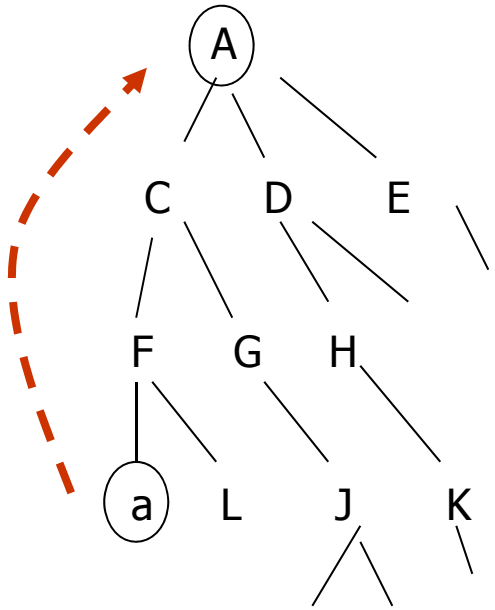
- For visse typer syntaksanalyse er det nødvendig å vite hvilke terminalsymboler som kan komme først i strenger som er avledet fra en ikketerminal A. Denne mengden kalles *First(A).*
  - F.eks. vil *First(if-setning)* i de fleste språk være bare nøkkelordet *"if"*, mens *First(tilordning)* ofte vil være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

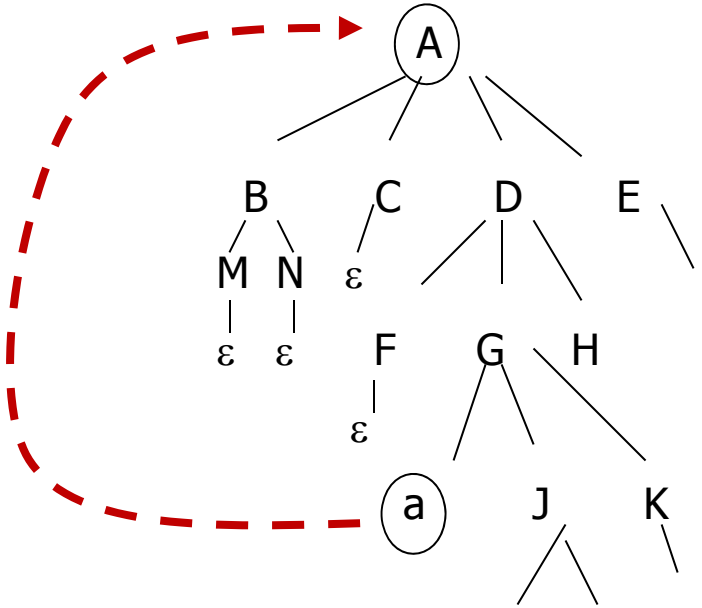
Enkel og kompliserte situasjoner: First-mengder

a ∈ First(A)



Angir informasjonsflyt under algoritmen

a ∈ First(A)



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

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( )*, se nederst.

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** }		

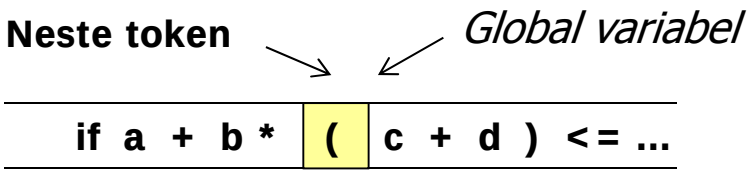
Kan fylle ut en tabell:

	First		
exp			**(** **n**
addop	+ −		
term		**(** **n**	
multop	*		
factor	**(** **n**		

NB: Kan godt ta reglene i vilkårlig rekke-følge, frem og tilbake til det "roer seg".

Eksempel: Rec. desc. av ren BNF

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



Hoved-idé:

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

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

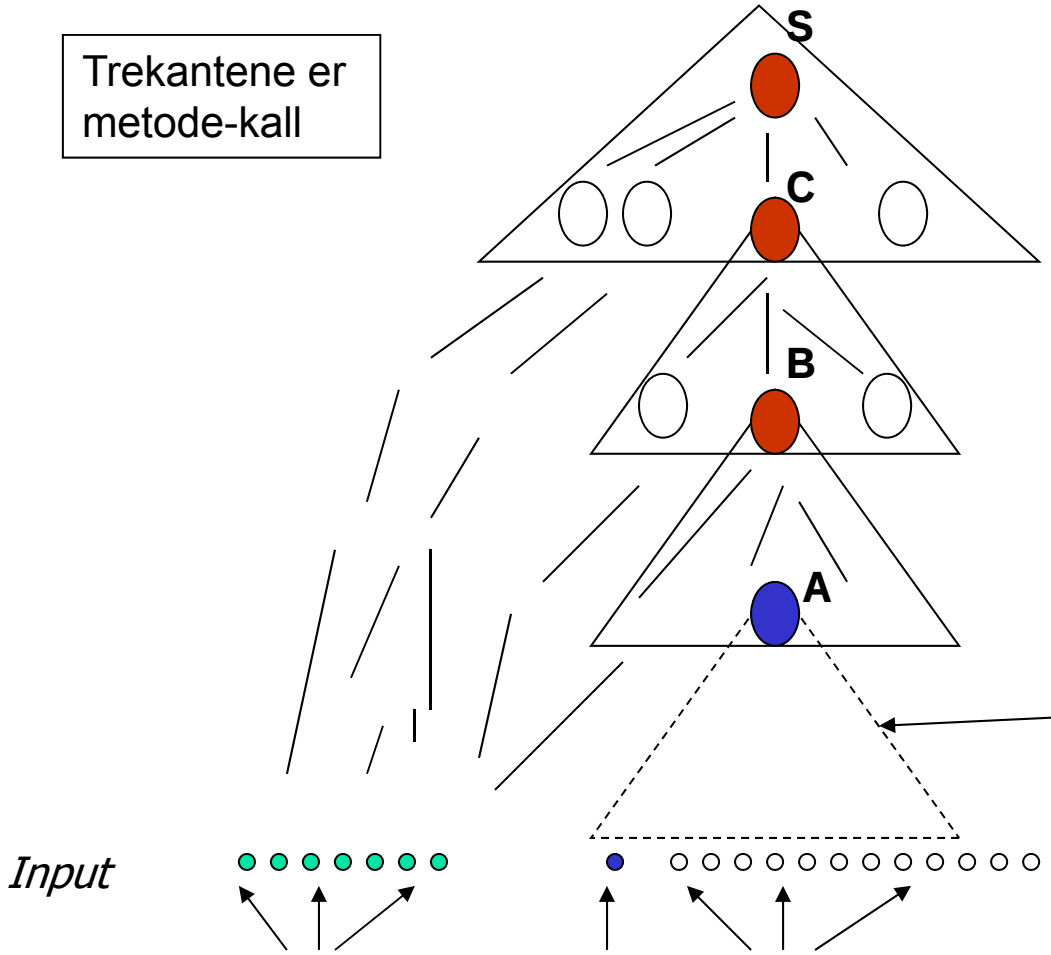
```
procedure factor ;
begin
  case token of
  ( :   match( ( ) ;
        exp ;
        match( ) ) ;
  number :
        match(number) ;
  else error ;
  end case ;
end factor ;
```

Sjekker at angitt terminal kommer, og om OK, leser inn neste. Brukes ofte for *bare* å lese (vet riktig tegn kommer). Da er det egentlig nok å kalle "getToken" (men i boka 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:
Metoden for en ikke-terminal (her A) kan alltid, ved hjelp av (bare) inneværende token, avgjøre hvilket alternativ som skal brukes for den aktuelle ikke-terminalen.

Vi må da formelt sett se på First-mengdene til hvert alternativ. Mer om det siden.

Hvilket alternativ for A?

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

Opprinnelig

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

Skrives ut som:

if-stmt → **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 ved ren BNF. Men ofte bedre med EBNF

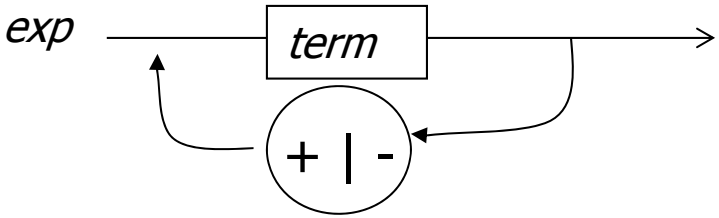
Gir lett uendelig mange rekursive kall ("rekursjons-brønn")

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

Bruker EBNF:

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

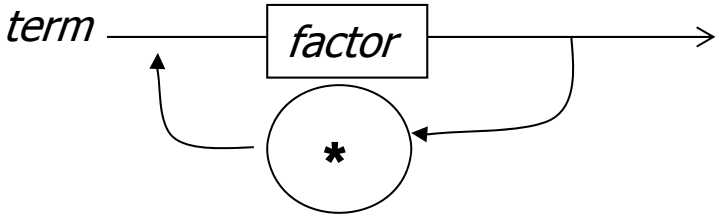
```
procedure exp ;
begin
   term ;
   while token = + or token = - do
      match (token) ;
      term ;
   end while ;
end exp ;
```



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

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

```
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

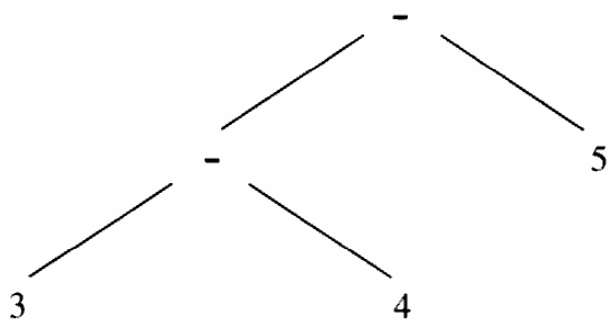
```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
   temp := term ;
   while token = + or token = - do
      case token of
       + : match (+) ;
           newtemp := makeOpNode(+) ;
           leftChild(newtemp) := temp ;
           rightChild(newtemp) := term ;
           temp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

NB: Kall

Alternativt:
newtemp.leftChild

NB: Kall

3-4-5:



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

Skriv prosedyre med trebygging for:

factor → (exp) | 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 → (exp) | number

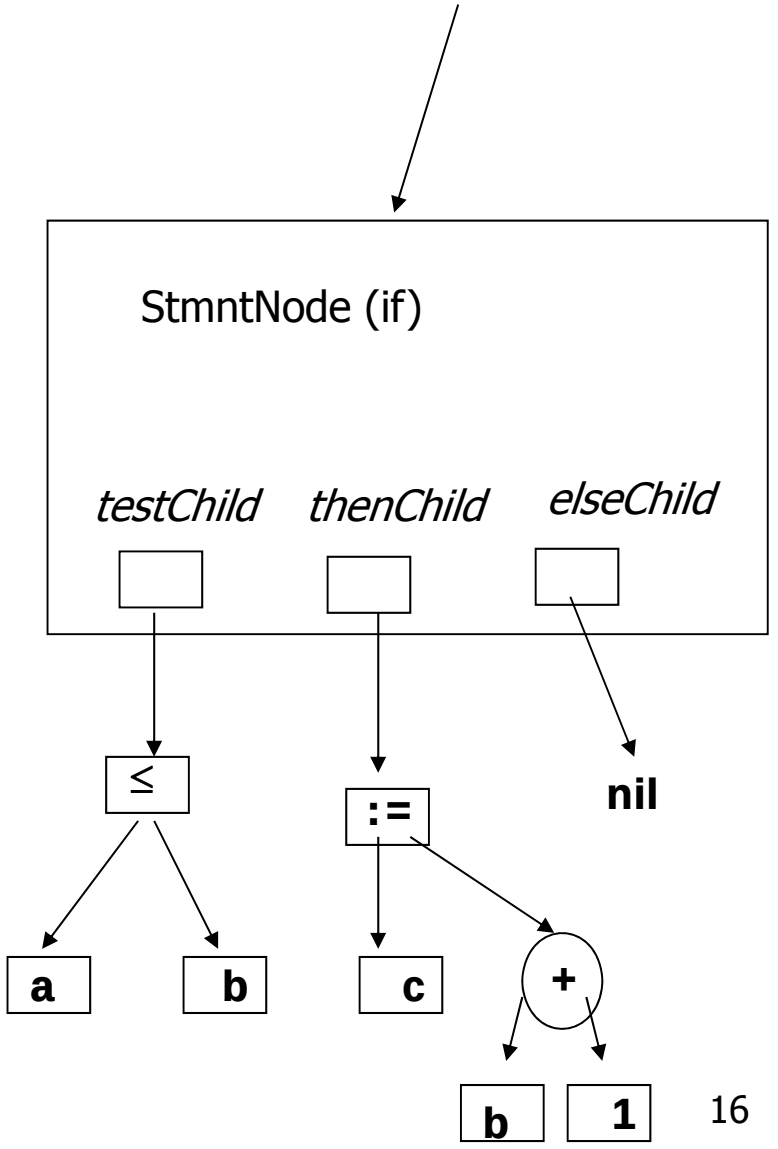
```
function factor: syntaxTree;
var fact: syntaxTree;
begin
    case token of
    ( :
          match ("(") ;
          fact = exp ;
          match (")") ;
    number :
          fact = makeNumberNode(number) ;
          match (number) ;
    else error(.....) ;
    end case;
    return fact;
end factor;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

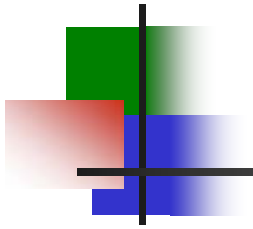
Gir "dummy-test"

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 ;
```





Et par greie transformasjoner

- For noen typer syntaksanalyse er det altså 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:

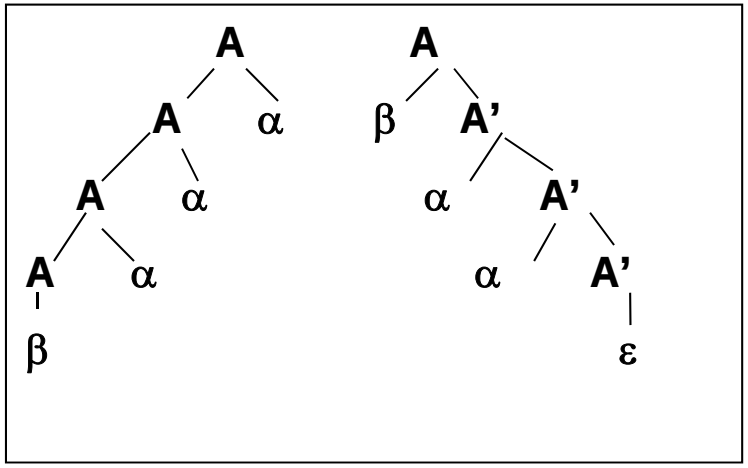
      if-setn $\rightarrow$ **if** (bet) setn-sekv **end** | **if** (bet) setn-sekv **else** setn-sekv **end**

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)*

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

Eks. side 160 – Fjerning av venstre-rekursjon

Den tradisjonelle entydige grammatikken

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

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

$\Downarrow$

$A \rightarrow \beta A'$

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

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

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

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**

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

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

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

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

---

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

Blir til:

if-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:

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

Avansert venstre-faktorisering. Ikke pensum!

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

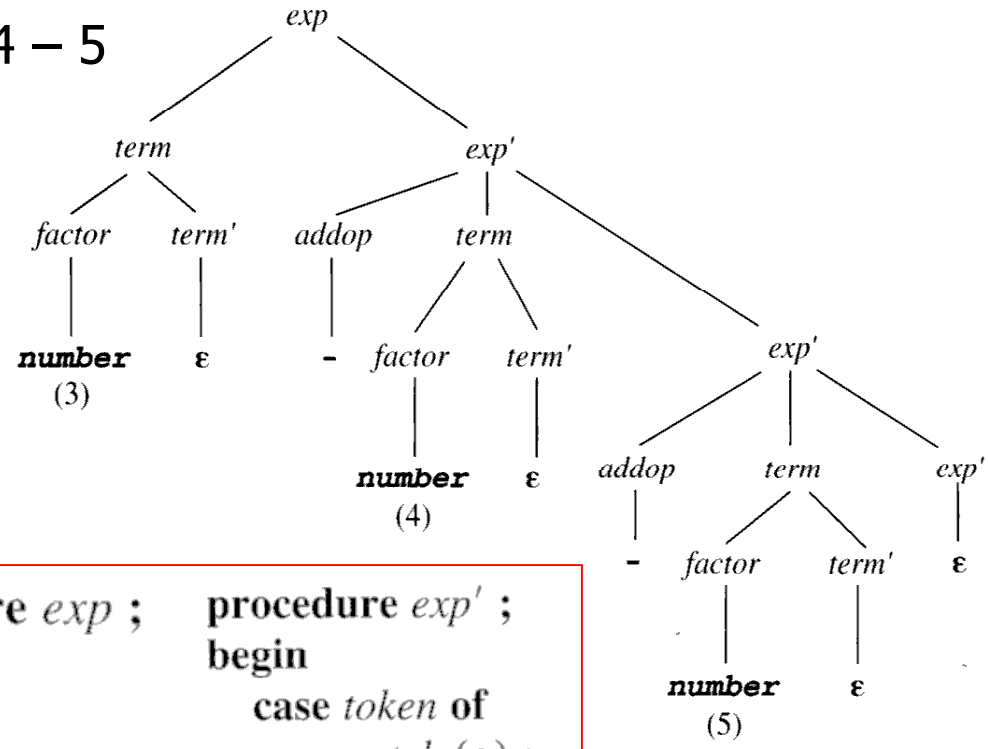
} Gjør som over. Pass på å få med alle alternativer med dette prefikset

Rec.decent etter tradisjonell fjerning av venstre-rekursjon

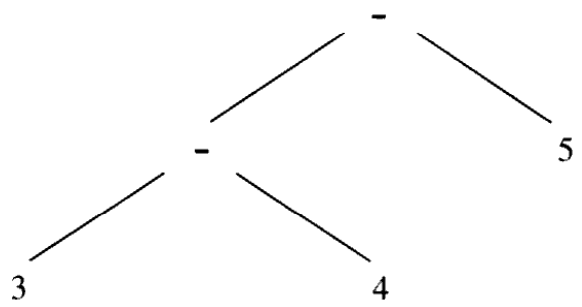
(Treet blir nå høyre assosiativt istedenfor venstre. Det må korrigeres for, se senere)

Uttrykk: 3 – 4 – 5

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



Det abstrakte syntakstreet vi Egentlig ønsket ønsket å lage:




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

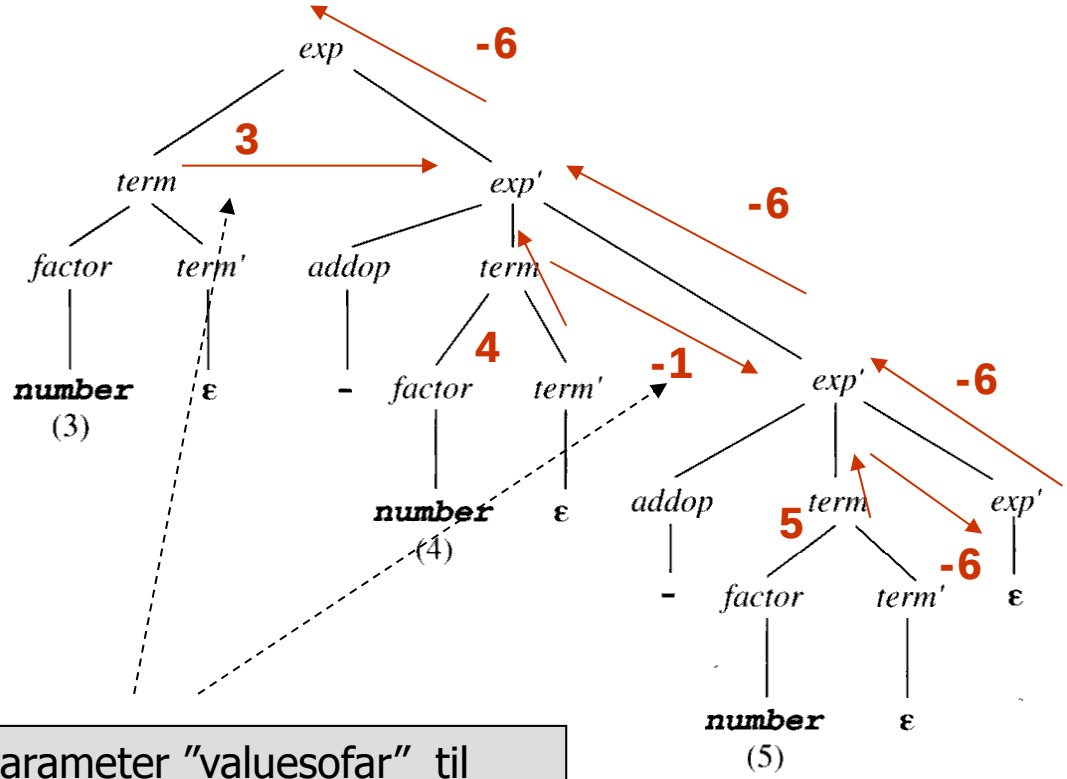
```
procedure exp' ;
begin
   case token of
    + : match (+) ;
        term ;
        exp' ;
    - : match (-) ;
        term ;
        exp' ;
   end case ;
end exp' ;
```

Rec.decent etter tradisjonell fjerning av venstre-rekursjon

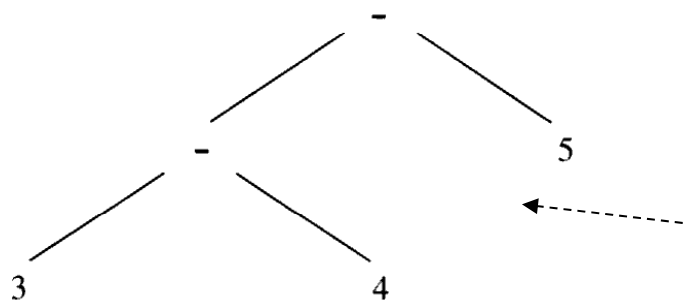
Treet er nå høyre assosiativt istedenfor venstre. Det må korrigeres for.

Lage tre eller beregne verdi : 3 – 4 – 5

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



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)

$exp \rightarrow term\ exp'$
$exp' \rightarrow addop\ term\ exp' \mid \varepsilon$
$addop \rightarrow$ **+** $\mid$ **-**
$term \rightarrow factor\ term'$
$term' \rightarrow mulop\ factor\ term' \mid \varepsilon$
$mulop \rightarrow$ *****
$factor \rightarrow$ **(** exp **)** $\mid$ **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

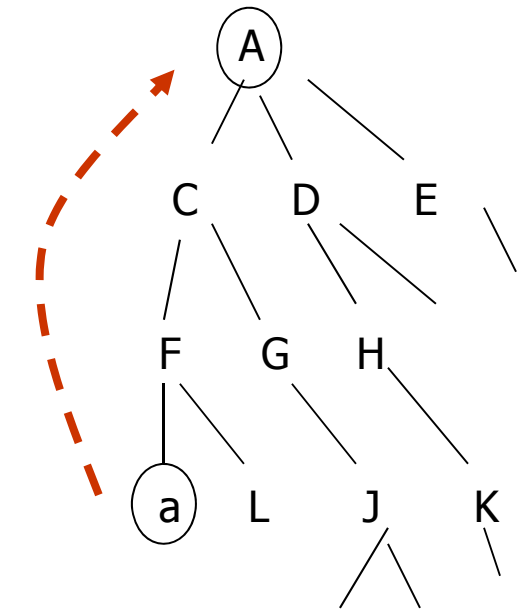
```
function exp' ( valsofar : integer ) : integer ;
begin
   if token = + or token = - then
      case token of
       + : match (+) ;
           valsofar := valsofar + term ;
       - : match (-) ;
           valsofar := valsofar - term ;
      end case ;
      return exp'(valsofar) ;
   else return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

ε -alternativet

Leverer verdien uendret oppover igjen.

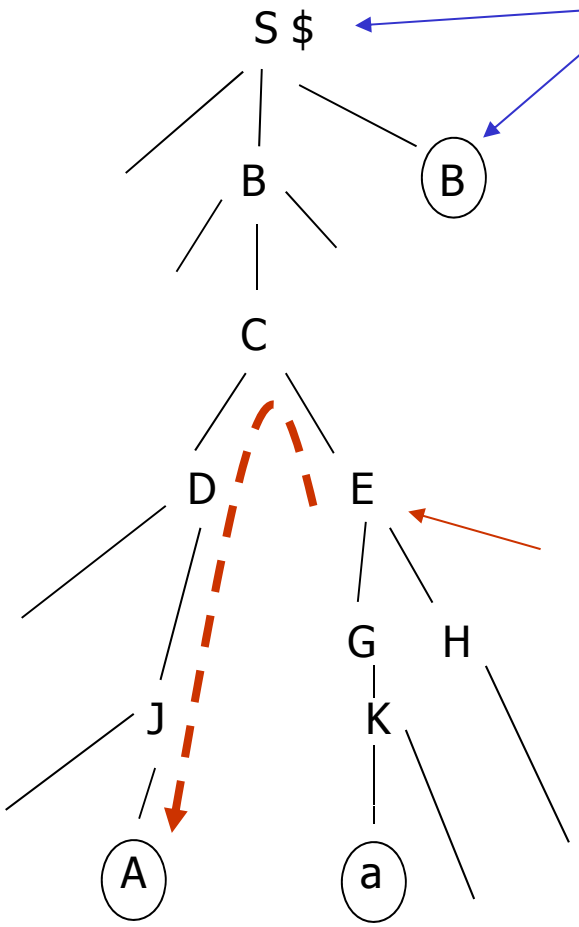
Vi har tatt First-mengder . Nå: Follow-mengder

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 å la Follow-mengden til B inneholde $.

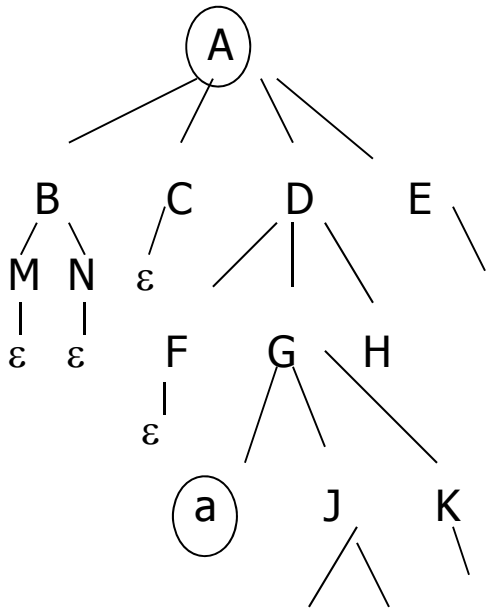
Terminal-symbolet $ tenkes altså å stå på slutten av enhver setning.

Vet fra før at
a ∈ **First(E)**

Merk: $ er alltid med i Follow-mengden til startsymbolet

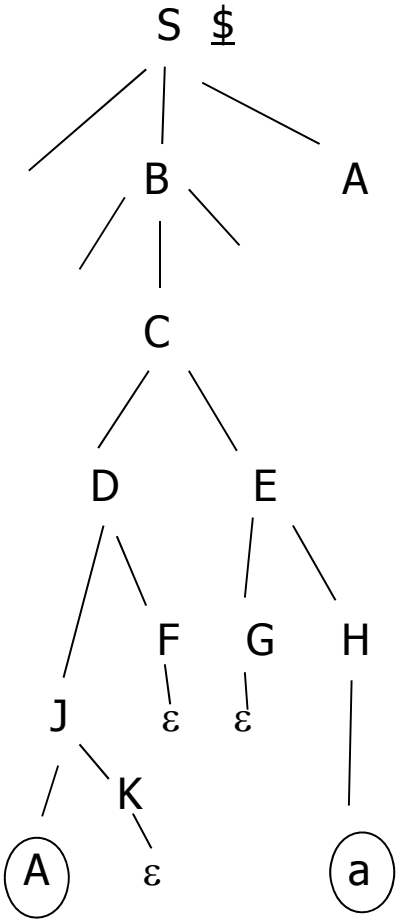
Kompliserte situasjoner: First- og Follow-mengder

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)



Beregning av Follow-mengder

β og γ er 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.

NB: Man må altså gjøre dette samlet for alle ikke-terminaler

Fra tidligere: 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 fylle ut en tabell:

	First		
exp			**(** **n**
addop	+ −		
term		**(** **n**	
multop	*		
factor	**(** **n**		

NB: Kan godt ta reglene i vilkårlig rekke-følge, frem og tilbake til det "roer seg".

Eksempel: 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*) = {$, **+**, **-**, *****, **)** }

Kan fylle ut en tabell ved å gå fram og tilbake i gram-matikken til det stopper:

	Follow	
exp	$ + –)	
addop	(**n**	
term	$ + – *	)
multop	(**n**	
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*) = {$, **+**, **-**, **)** }	