INF5110 – V2013

Stoff som i boka står i kap 4, men som er generelt stoff om grammatikker

29. januar 2013

Stein Krogdahl, Ifi, UiO

NB: Ikke undervisning fredag 1. februar!

Oppgaver som gjennomgås 5. februar på neste sider

Oppgaver som gjennomgås 5. februar:

- Spørsmålene på de to neste foilene
- Finn *First*- og *Follow* -mengdene til grammatikken i eksempelet på side 160 *etter at* venstre-rekursjon er fjernet.
- Beskriv en algoritme som bare finner de utnullbare ikke-terminaler, uten å beregne First og Follow.

Noen oppgaver som gjennomgås 5. februar

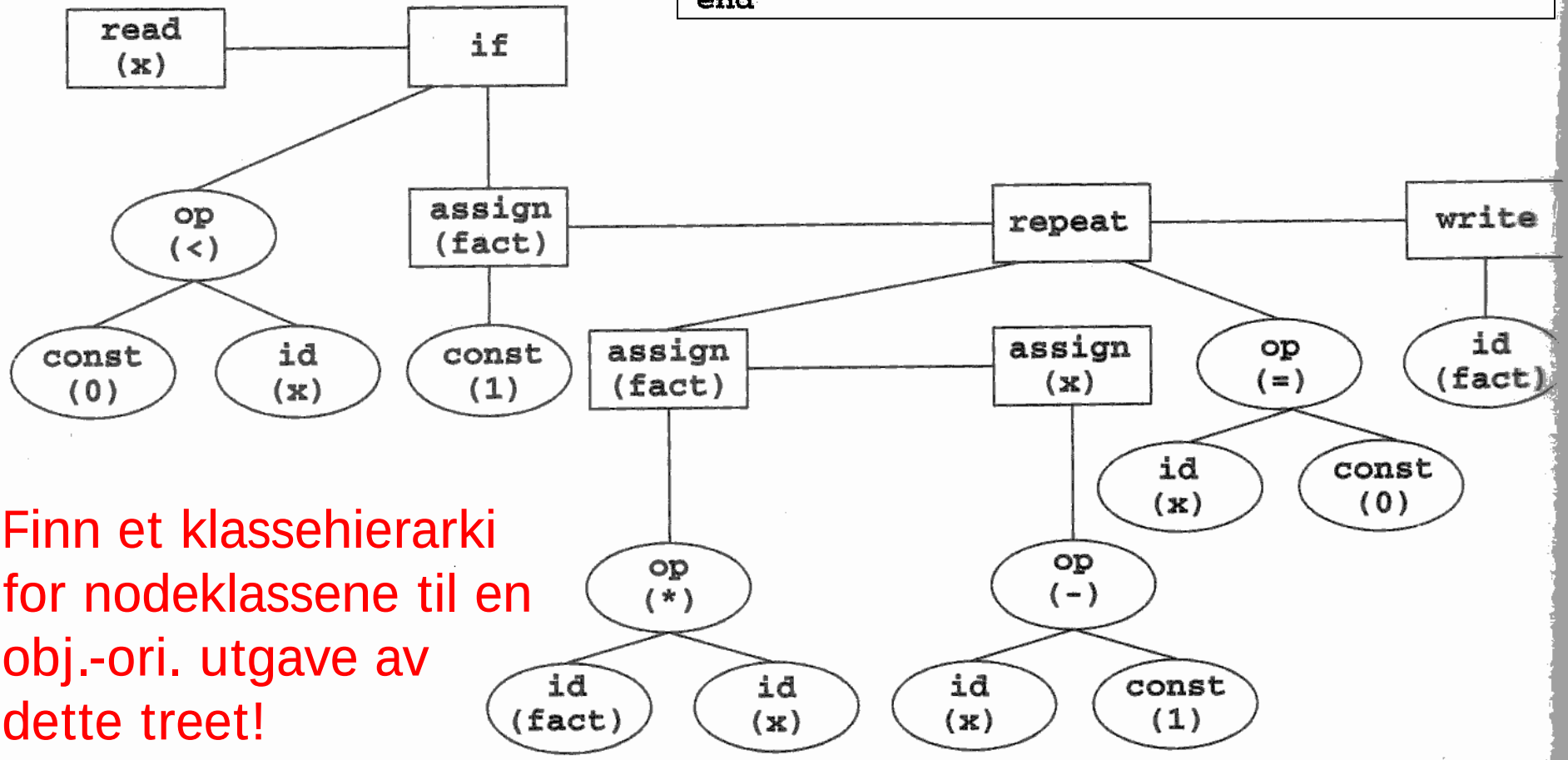
program → *stmt-sequence*
stmt-sequence → *stmt-sequence* **;** *statement* | *statement*
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *exp* **then** *stmt-sequence* **end**
| **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**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

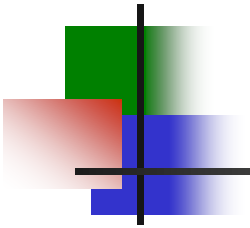
Oppgave til 5. februar

```
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!



Dagens temaer

- Noe avsluttende fra sist
- First og Follow-mengder
  - Boka ser på en av parseringsmetodene først, uten å se på First/Follow-mengder.
  - Vi tar dette først (hører naturlig til i kap 3)
- To greie transformasjoner på grammatikker
  - Også naturlig i kap 3
  - Fjerning av venstre-rekursjon
  - Venstre-faktorisering
- Det blir altså i dag en del ”detaljert pirk”
  - ... uten at vi nå får se anvendelser av det

First og Follow-mengder, hvorfor?

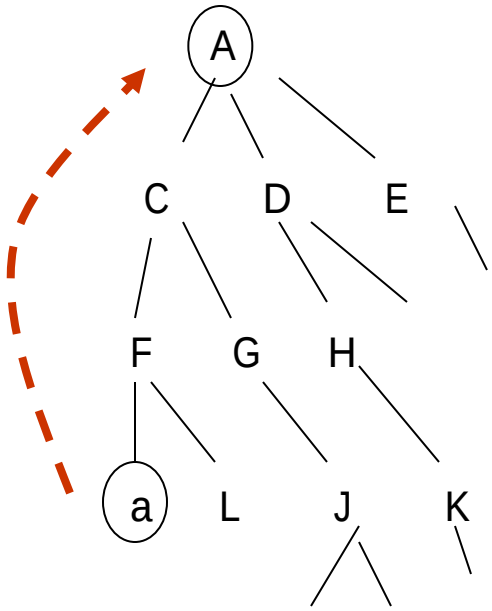
- For visse typer syntaksanalyse er det nødvendig å vite hvilke terminalsymboler som kan komme først i strenger som er avledet fra en ikketerminal A. Denne mengden kalles *First(A).*
  - F.eks. vil *First(if-setning)* i de fleste språk være bare nøkkelordet *"if"*, mens *First(tilordning)* ofte vil være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

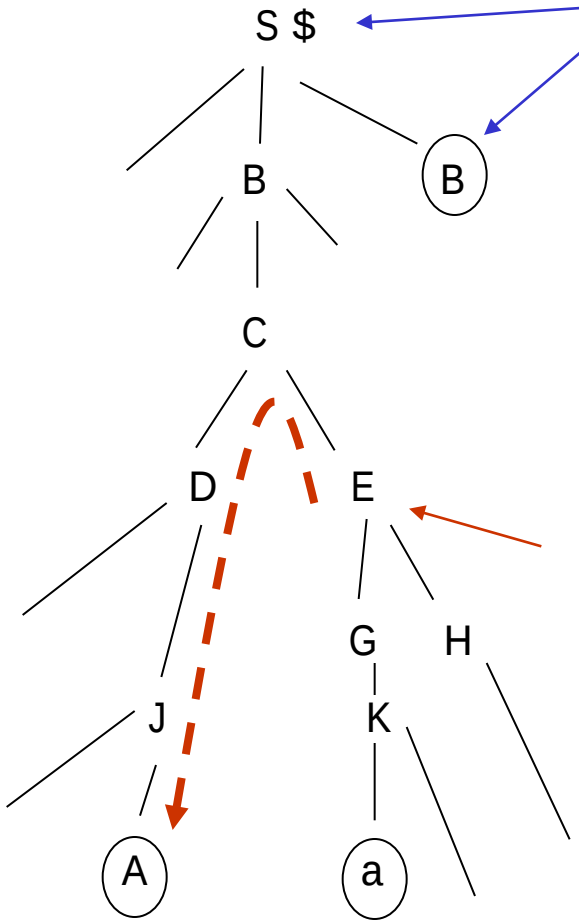
Typiske situasjoner: First- og Follow-mengder

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 å la Follow-mengden til B inneholde $.

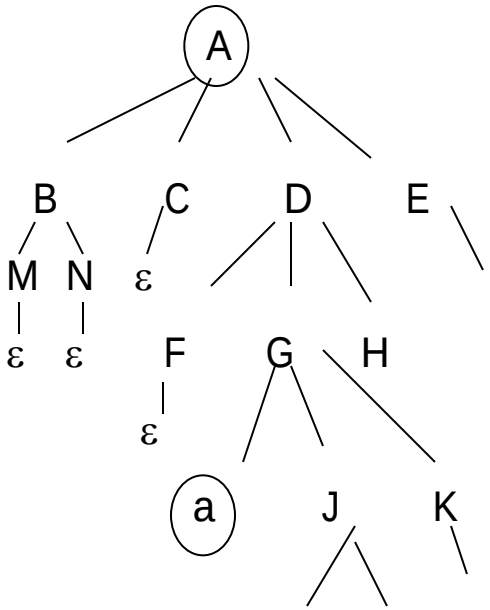
Terminal-symbolet $ tenkes altså å stå på slutten av enhver setning.

Vet fra før at
a ∈ **First(E)**

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

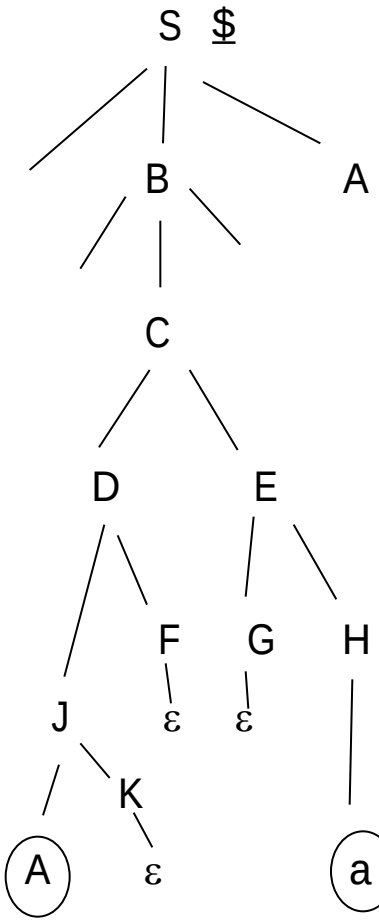
Kompliserte situasjoner: First- og Follow-mengder

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)



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”.

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)
```

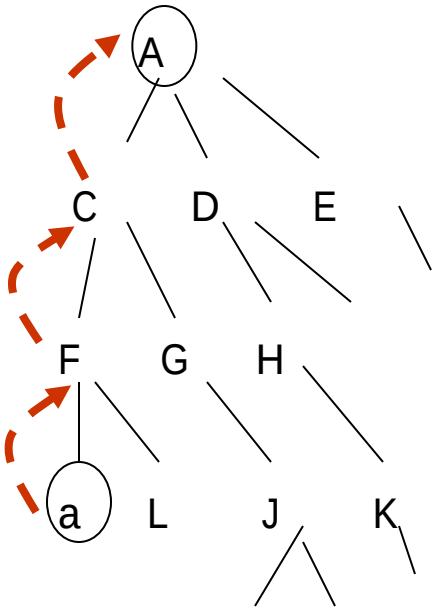
Algoritme:

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

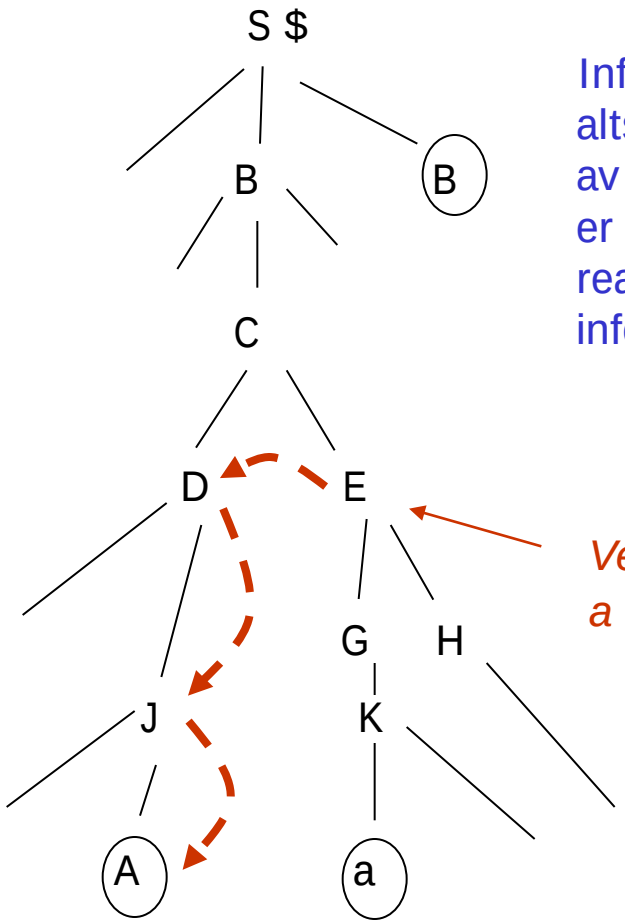
NB: Man må altså gjøre dette samlet for alle ikke-terminaler

Bedre bilde av informasjonsflyten: First- og Follow-mengder

a ∈ First(A)



a ∈ Follow(A)



Infomasjonen går altså bare ett «hakk» av gangen, så dette er vel et mer realistisk bilde av informasjonsflyten

Vet fra før at
a ∈ **First(E)**

Angir informasjonsflyt under algoritmene

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

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:

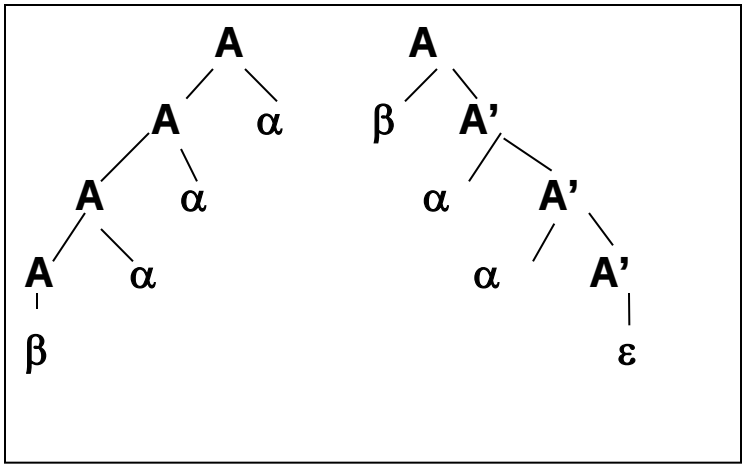
      if-setn $\rightarrow$ **if** (bet) setn-sekv **end** | **if** (bet) setn-sekv **else** setn-sekv **end**

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

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

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

$\Downarrow$

$A \rightarrow \beta A'$

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

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

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

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*) *setn* *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

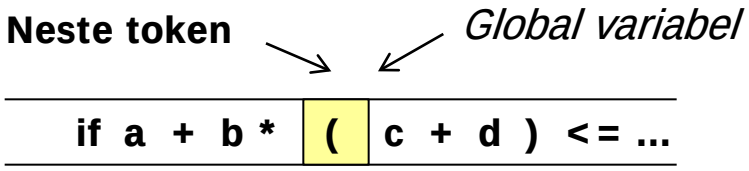
Kap. 4: Parsering ovenfra-ned (top-down)

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: «Recursive descent» 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, som blir veldig enkel.

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

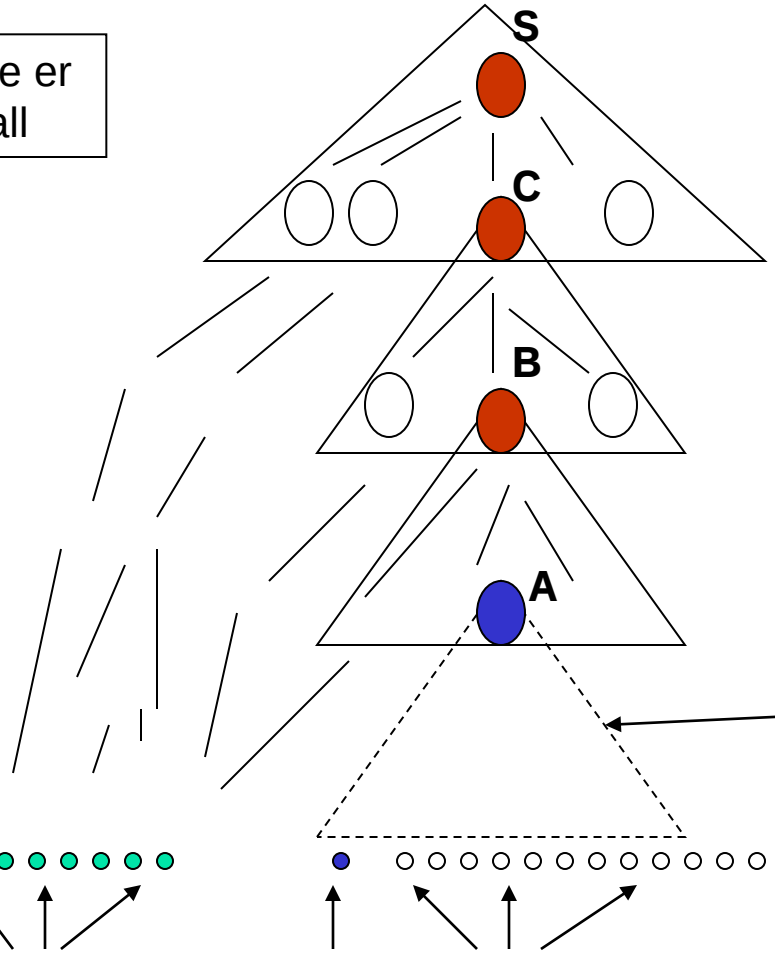
Sjekker at angitt terminal kommer, og ”leser til neste”. Brukes ofte *bare* for å lese (sjekken må slå til). Da er det egentlig nok å kalle ”getToken” (men her kalles alltid ”match”)

```
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) førstkommende 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

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