INF5110 – V2010
Hovedstoffet i kap 4:
Parsering ovenfra-ned (”top down”)

16. Februar 2010

Stein Krogdahl

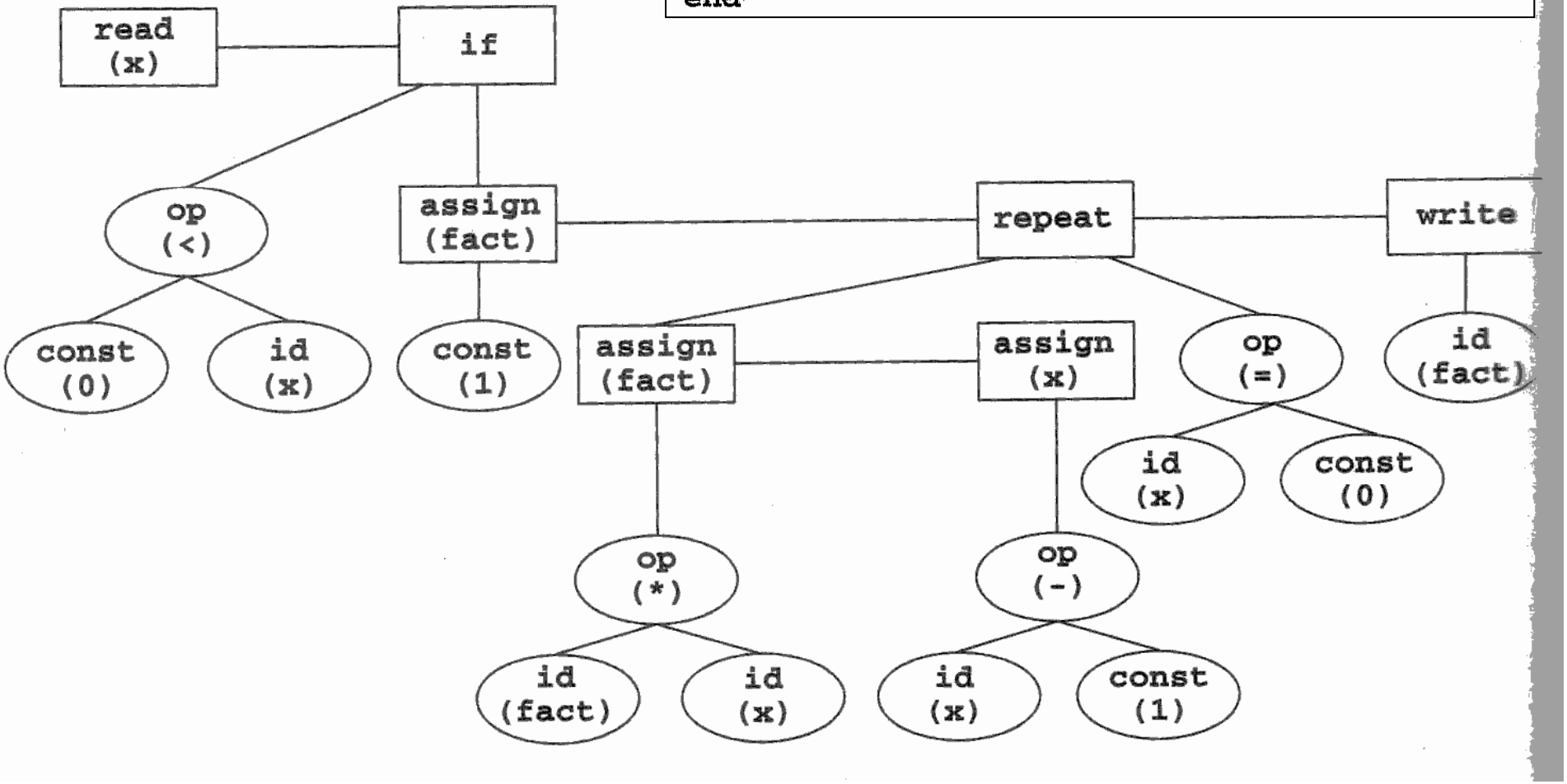
Ifi, UiO

Oppgaver som gjennomgås i dag:

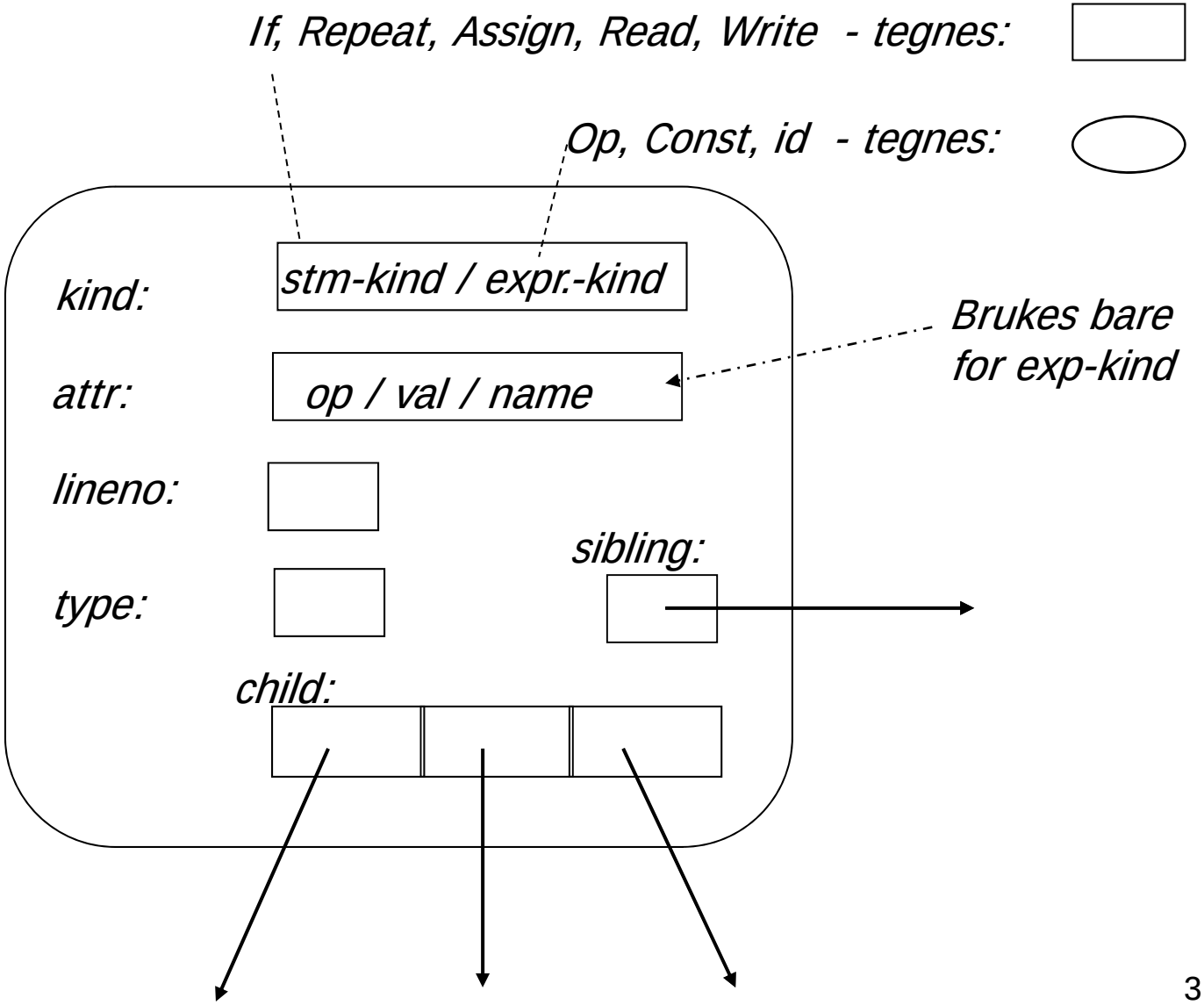
- Spørsmålene på foil 35 og 36 fra 9/2
- Finn First og Follow til grammatikken nederst på foil 15 fra 10/2

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
```



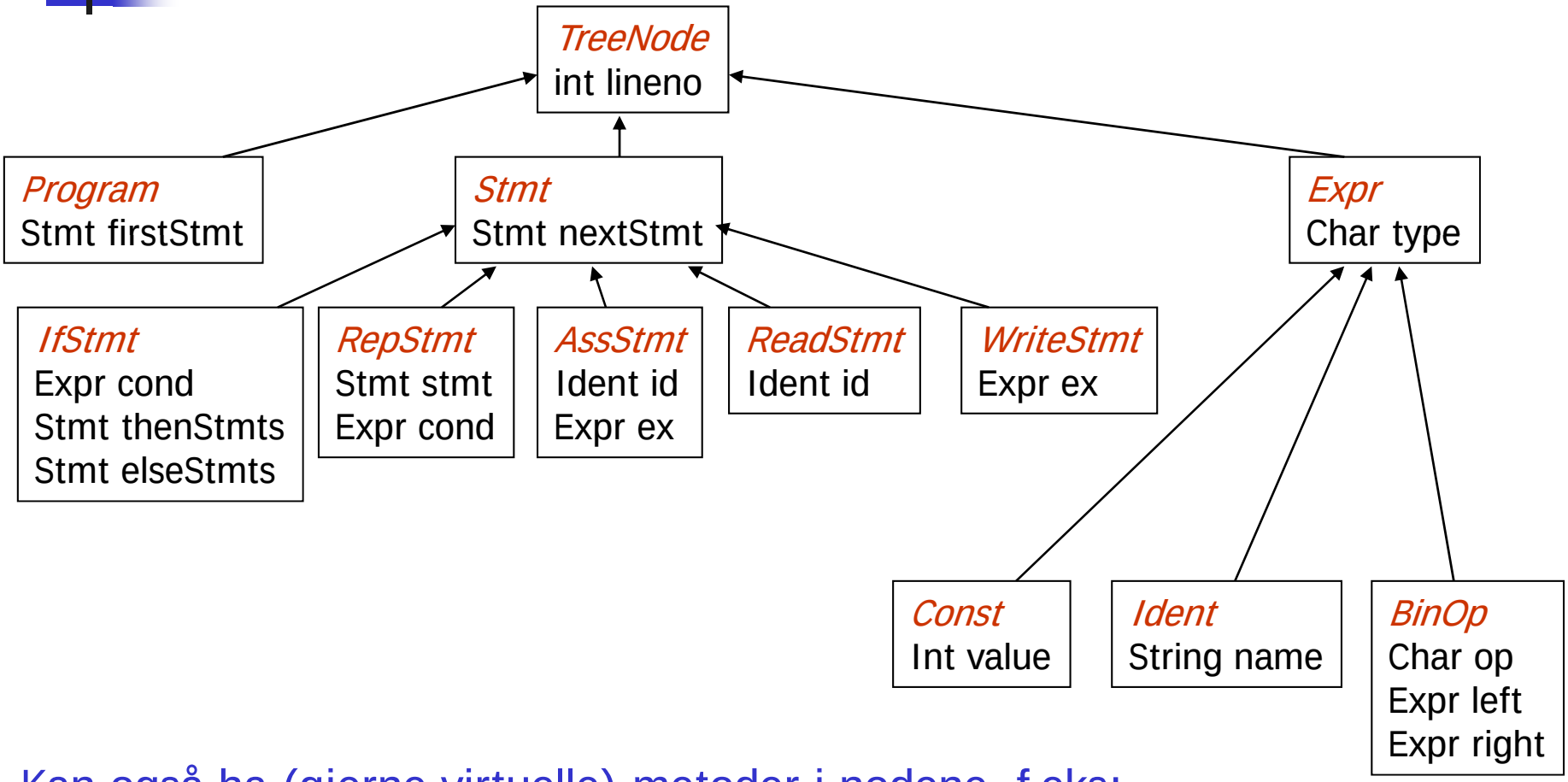
Nodestruktur i C for Tiny



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

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



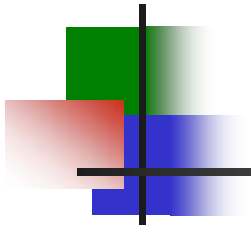
Kan også ha (gjerne virtuelle) metoder i nodene, f.eks:

- 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

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 - > stmt-sequence statement ; | *statement ;*
- Hva slags assosiativitet 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

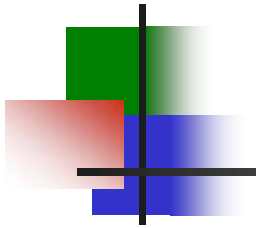
First og follow for transformert uttr.gram.

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:

$$
\begin{aligned}
\text{First}(exp) &= \{\texttt{(}, \textbf{\texttt{number}}\} & \text{Follow}(exp) &= \{\$, \texttt{)}\} \\
\text{First}(exp') &= \{\texttt{+}, \texttt{-}, \varepsilon\} & \text{Follow}(exp') &= \{\$, \texttt{)}\} \\
\text{First}(addop) &= \{\texttt{+}, \texttt{-}\} & \text{Follow}(addop) &= \{\texttt{(}, \textbf{\texttt{number}}\} \\
\text{First}(term) &= \{\texttt{(}, \textbf{\texttt{number}}\} & \text{Follow}(term) &= \{\$, \texttt{)}, \texttt{+}, \texttt{-}\} \\
\text{First}(term') &= \{\texttt{*}, \varepsilon\} & \text{Follow}(term') &= \{\$, \texttt{)}, \texttt{+}, \texttt{-}\} \\
\text{First}(mulop) &= \{\texttt{*}\} & \text{Follow}(mulop) &= \{\texttt{(}, \textbf{\texttt{number}}\} \\
\text{First}(factor) &= \{\texttt{(}, \textbf{\texttt{number}}\} & \text{Follow}(factor) &= \{\$, \texttt{)}, \texttt{+}, \texttt{-}, \texttt{*}\}
\end{aligned}
$$



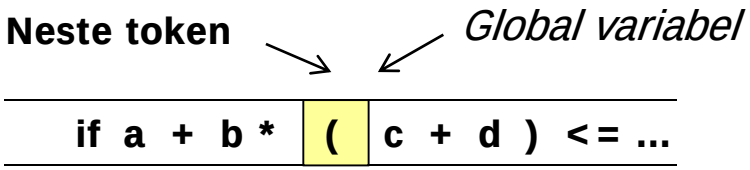
Kap. 4: Ovenfra-ned (top-down) parsering

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 i år.
  - *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: En ren BNF-grammatikk som tilfredstiller dette kravet.

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, og parsere 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 ;
```

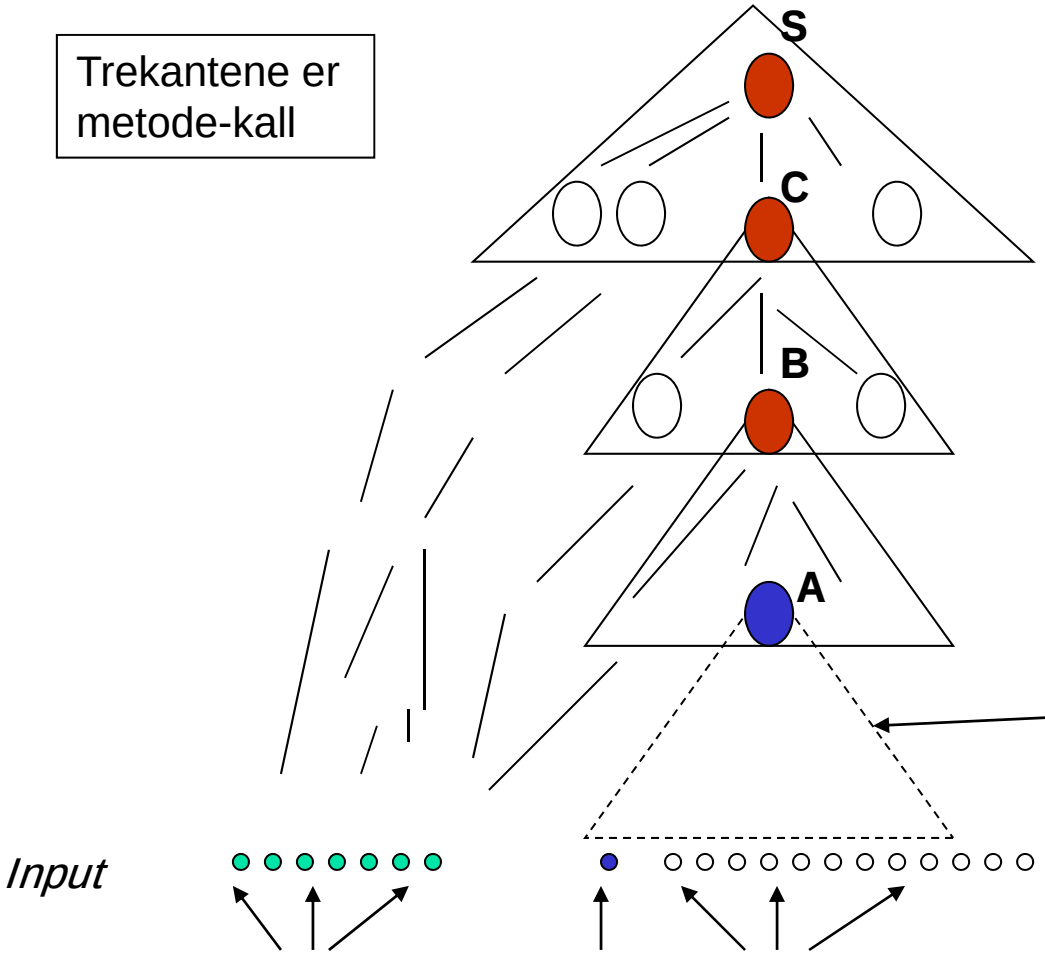
Sjekker at angitt terminal kommer, og "leser til neste". Brukes ofte *bare* for å lese (sjekken må slå til). Da er det egentlig nok å kalle "getToken" (men her kalles alltid "match")

```
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) neste token, avgjøre hvilket alternativ som skal brukes

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. Går ofte med EBNF

Gir uendelig mange rekursive kall

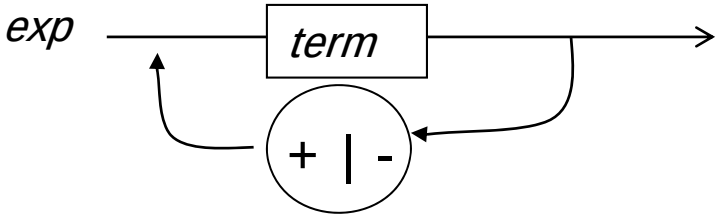
$exp \rightarrow exp\ addop\ term \mid term$

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

Bruker EBNF:

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

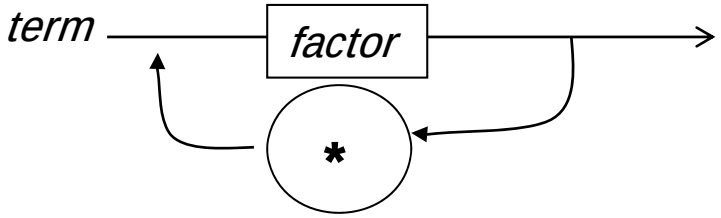
```
procedure exp ;
begin
   term ;
   while token = + or token = - do
      match (token) ;
      term ;
   end while ;
end exp ;
```



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

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

```
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

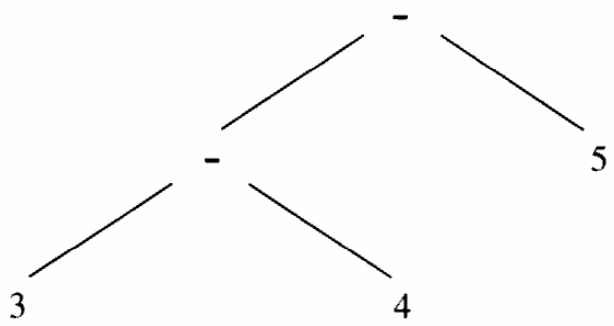
```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
   temp := term ;
   while token = + or token = - do
      case token of
       + : match (+) ;
           newtemp := makeOpNode(+) ;
           leftChild(newtemp) := temp ;
           rightChild(newtemp) := term ;
           temp := 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 én "term", så lages ingen ny node. Vi leverer den vi har fått

Skriv prosedyre med trebygging for:

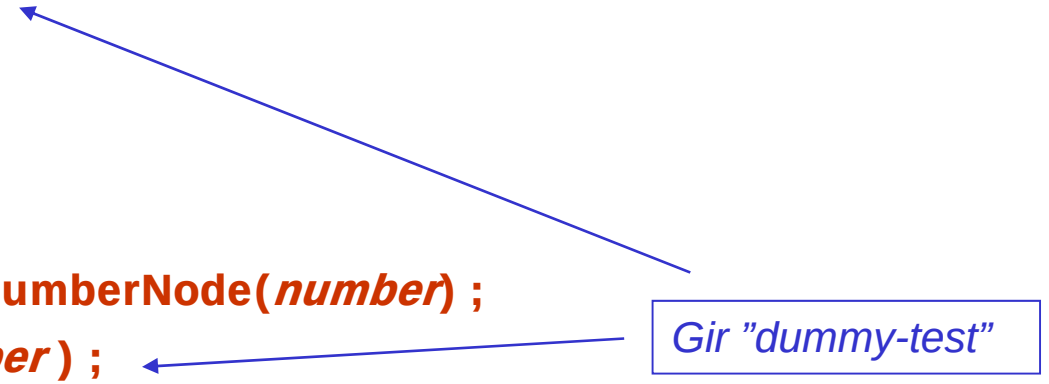
factor → *(exp)* | *number*

```
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*

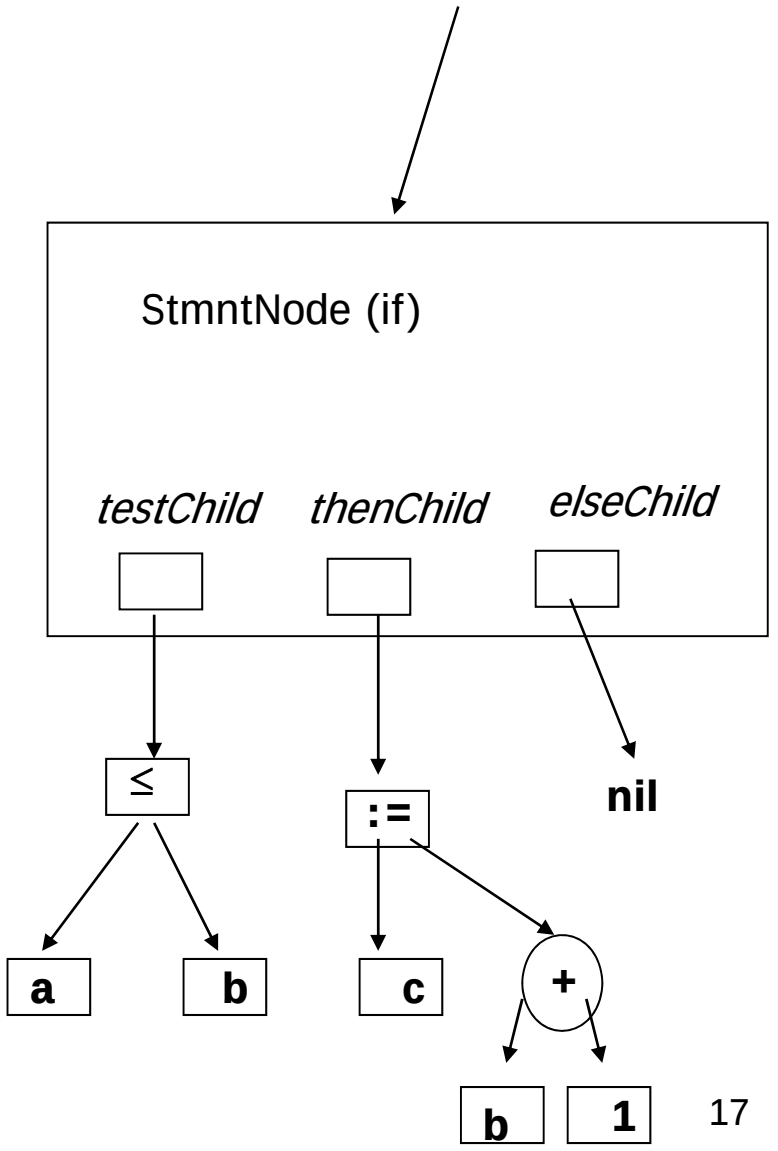
```
proc factor : syntaxTree;
var fact: syntaxTree;
begin
    case token of
    ( :
        match ("(") ;
        fact = exp ;
        match (")") ;
    number :
        fact = makeNumberNode(number) ;
        match (number) ;
    else error(.....) ;
    end case
    return fact;
end factor;
```



Parsering av if-setning, med tre-generering

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

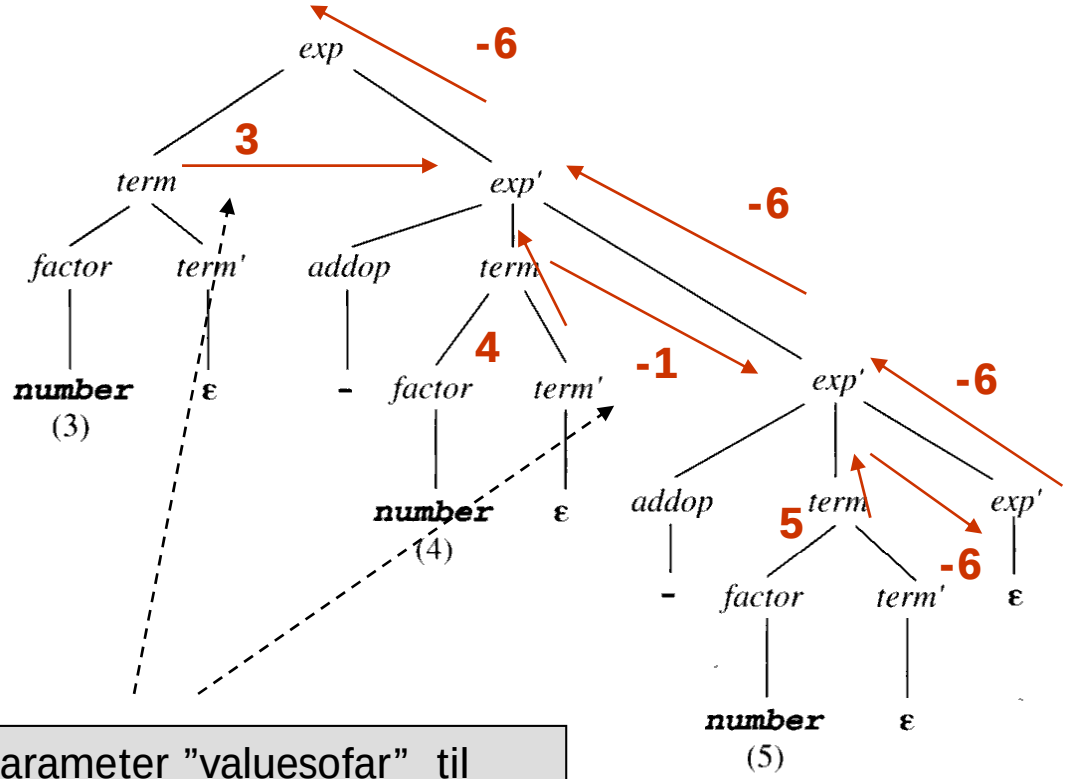
```
function ifStatement : syntaxTree ;
var temp : syntaxTree ;
begin
   match (if) ;
   match ( ( ) ;
   temp := makeStmtNode(if) ;
   testChild(temp) := exp ;
   match ( ) ) ;
   thenChild(temp) := statement ;
   if token = else then
      match (else) ;
      elseChild(temp) := statement ;
   else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



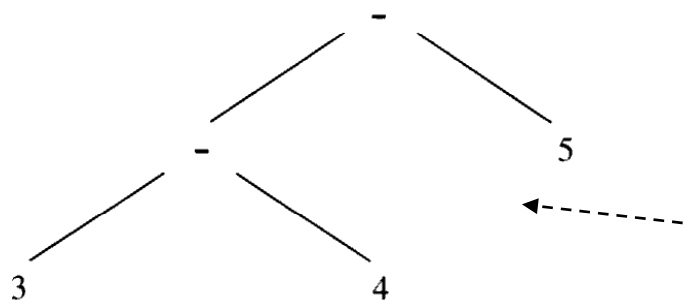
Rec.decent etter tradisjonell fjerning av venstre-rekursjon (treet er nå høyre assosiativt istedenfor venstre). Det må korrigeres for.

Lage tre eller beregne verdi : 3 – 4 – 5

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



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)

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

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 ∈ First(α)

2. Altså, dersom:

- α er utnullbar, og
- a ∈ 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

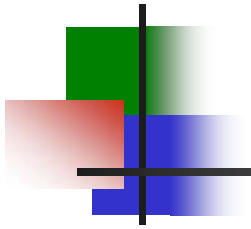
Har fjernet venstre-rekursjon:

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

Vi får da følgende First- og Follow-mengder:

$$
\begin{aligned}
\text{First}(exp) &= \{\texttt{(}, \textbf{\texttt{number}}\} & \text{Follow}(exp) &= \{\$, \texttt{)}\} \\
\text{First}(exp') &= \{\texttt{+}, \texttt{-}, \varepsilon\} & \text{Follow}(exp') &= \{\$, \texttt{)}\} \\
\text{First}(addop) &= \{\texttt{+}, \texttt{-}\} & \text{Follow}(addop) &= \{\texttt{(}, \textbf{\texttt{number}}\} \\
\text{First}(term) &= \{\texttt{(}, \textbf{\texttt{number}}\} & \text{Follow}(term) &= \{\$, \texttt{)}, \texttt{+}, \texttt{-}\} \\
\text{First}(term') &= \{\texttt{*}, \varepsilon\} & \text{Follow}(term') &= \{\$, \texttt{)}, \texttt{+}, \texttt{-}\} \\
\text{First}(mulop) &= \{\texttt{*}\} & \text{Follow}(mulop) &= \{\texttt{(}, \textbf{\texttt{number}}\} \\
\text{First}(factor) &= \{\texttt{(}, \textbf{\texttt{number}}\} & \text{Follow}(factor) &= \{\$, \texttt{)}, \texttt{+}, \texttt{-}, \texttt{*}\}
\end{aligned}
$$

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**					



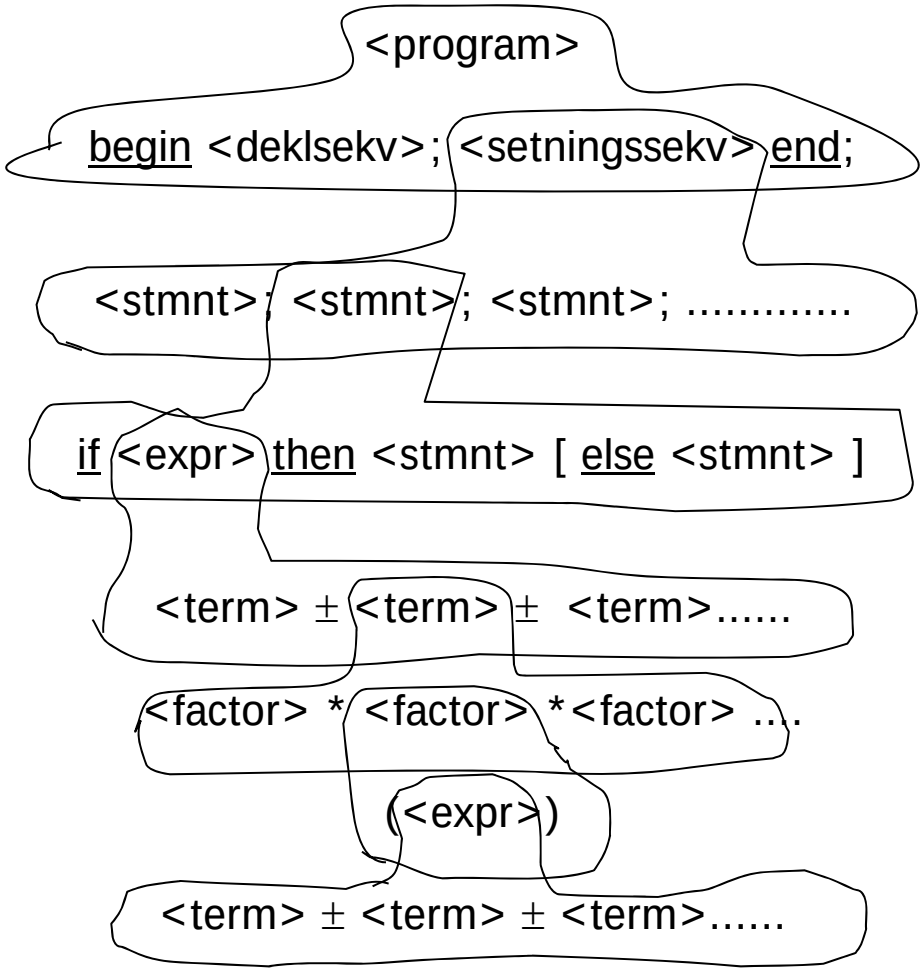
Når kompilatoren oppdager feil

- 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 { +, - } ?

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(…)

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

```
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 ∪ { $ }) do
     getToken ;
end scanto ;

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