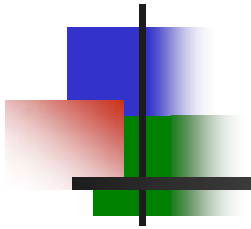
INF5110, onsdag 8. februar 2012

Mer fra kap 4:

Parsering ovenfra-ned (”top down”)

Stein Krogdahl Ifi, UiO

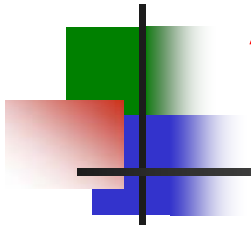


Merk: Ikke forelesning på onsdag 15. februar

Oppgaver som gjennomgås i dag:

(Svar ligger bakerst. Ikke titt!)

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



Annonse:
Søk jobb på ”Package Template”-kompilatoren!

- I et prosjekt (SWAT-prosjektet) har vi designet en språkmekanisme som kalles ”Package Templates”, forkortet PT
- Vi har beskrevet hvordan denne mekanismen kan legges inn som en utvidelse av Java, til et språk ”JPT”
- Vi er nå kommet langt med en kompilator for JPT, og mye er allerede på plass.
- Vi hadde to til å jobbe på kompilatoren, men nå har vi bare en (Steinar Kaldager)
- Vi trenger en til, bl.a. til å jobbe med uttesting.
- Lønn: ca. 170 kr/time ?
- Søknadsfrist: fredag 10/2, til steinkr@ifi.uio.no
- Bare send en mail med relevante opplysninger (også karakterer i relevante fag)

LL(1) – grammatikk

1. Altså, dersom

   a ∈ First(α)

så legg

A -> α inn i M[A, a]

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

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

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

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

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

2. Altså, dersom:

- α er utnullbar, og
- a ∈ Follow(A)

så legg A -> α inn i M[A, a]

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: Selv om man

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

For tabellen ble ikke entydig her

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

NB: Ikke i boka, men *er* pensum!

- Den ”utvidede startmengden”, *Efirst*, til en produksjon det som kan ligge som førstkommende token i input dersom **A** $\rightarrow \alpha$ et riktig valg på dette stadiet under parseringen.
- Her må man også tenke på tilfellet at α kan være utnullbar, og da kan også Follow(**A**) komme som førstkommende token
- Mengden kan beregnes slik:

  Efirst (**A** $\rightarrow \alpha$) = First(α), pluss, om α er utnullbar, Follow(**A**).
- Vi ser da at produksjonen **A** $\rightarrow \alpha$ skal inn i M[A, a] hvis og bare hvis a $\in$ Efirst (**A** $\rightarrow \alpha$)
- Dermed, får vi en alternativ definisjon av LL(1):

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

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

På forrige side vil da både

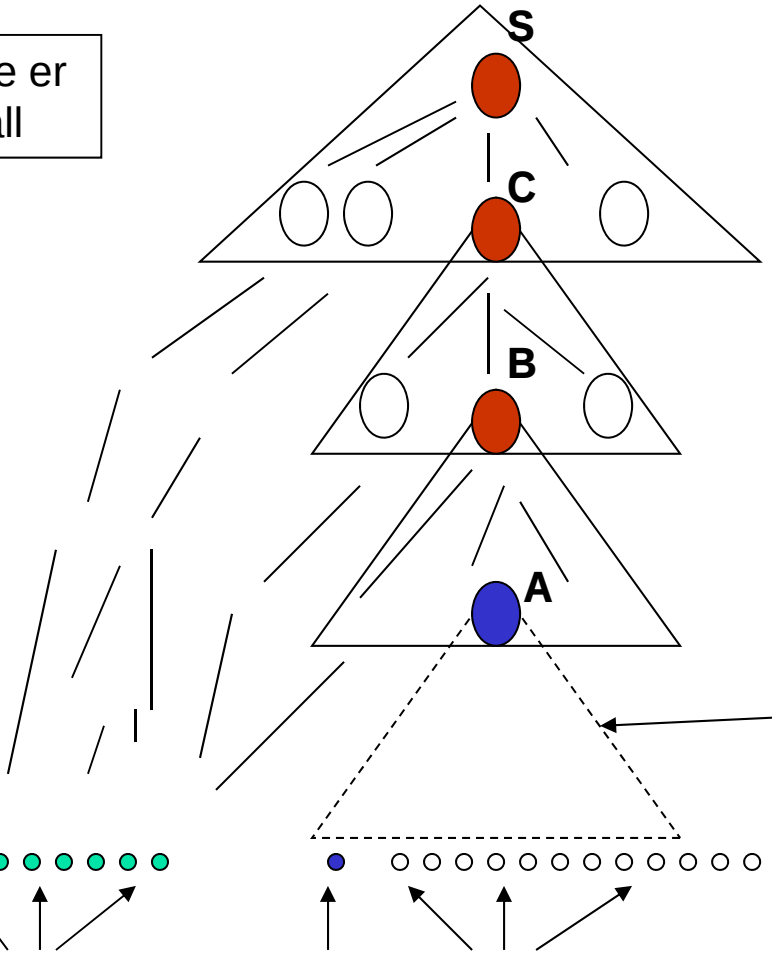
Efirst(else-part -> **else** *statmt)*
og
Efirst(else-part -> ε)

inneholde **else**
Dermed er grammatikken *ikke* LL(1)

Tidligere foil:

Situasjonen under rekursiv 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)

Hvilket alternativ for A?

Input

Analysert og funnet OK **token** Skal matches mot resten av det valgte alternativ for A

LL(1) –tabell for uttrykks-grammatikk

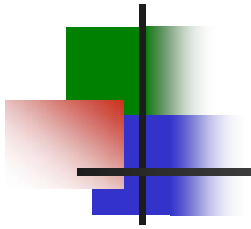
Har fjernet venstre-rekursjon:

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

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**					



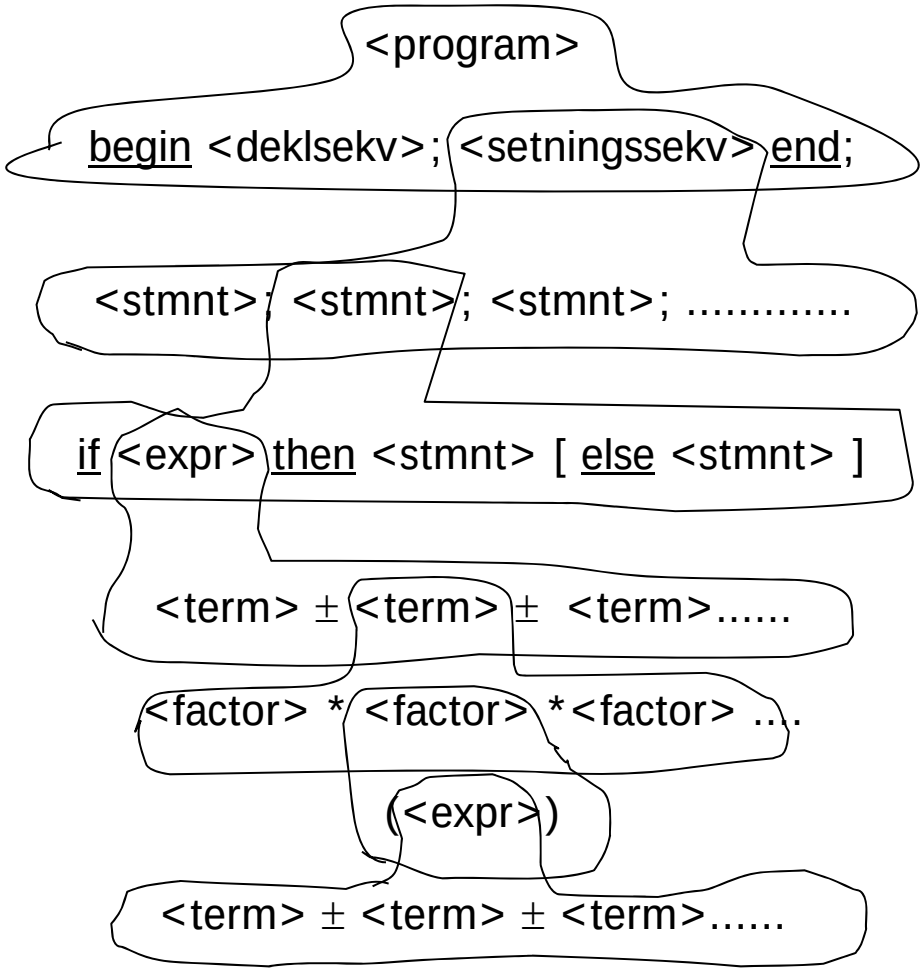
Når kompilatoren oppdager feil

- 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



Synch-set (stakk eller parameter):

end

; First(stmt)

navn if while for ...

then First(stmt) else

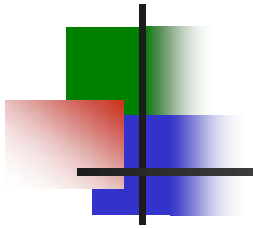
+ - First(term)

(tall navn

* First(factor)

)

+ - (tall navn



Syntaksfeil ved ”rec. descent”

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”?

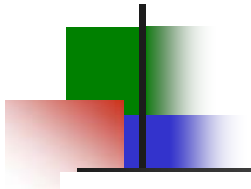
```
procedure scanto ( synchset ) ;
begin
  while not ( token in synchset ∪ { $ }) do
     getToken ;
end scanto ;

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

Løsningsforslag på oppgaver

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?*



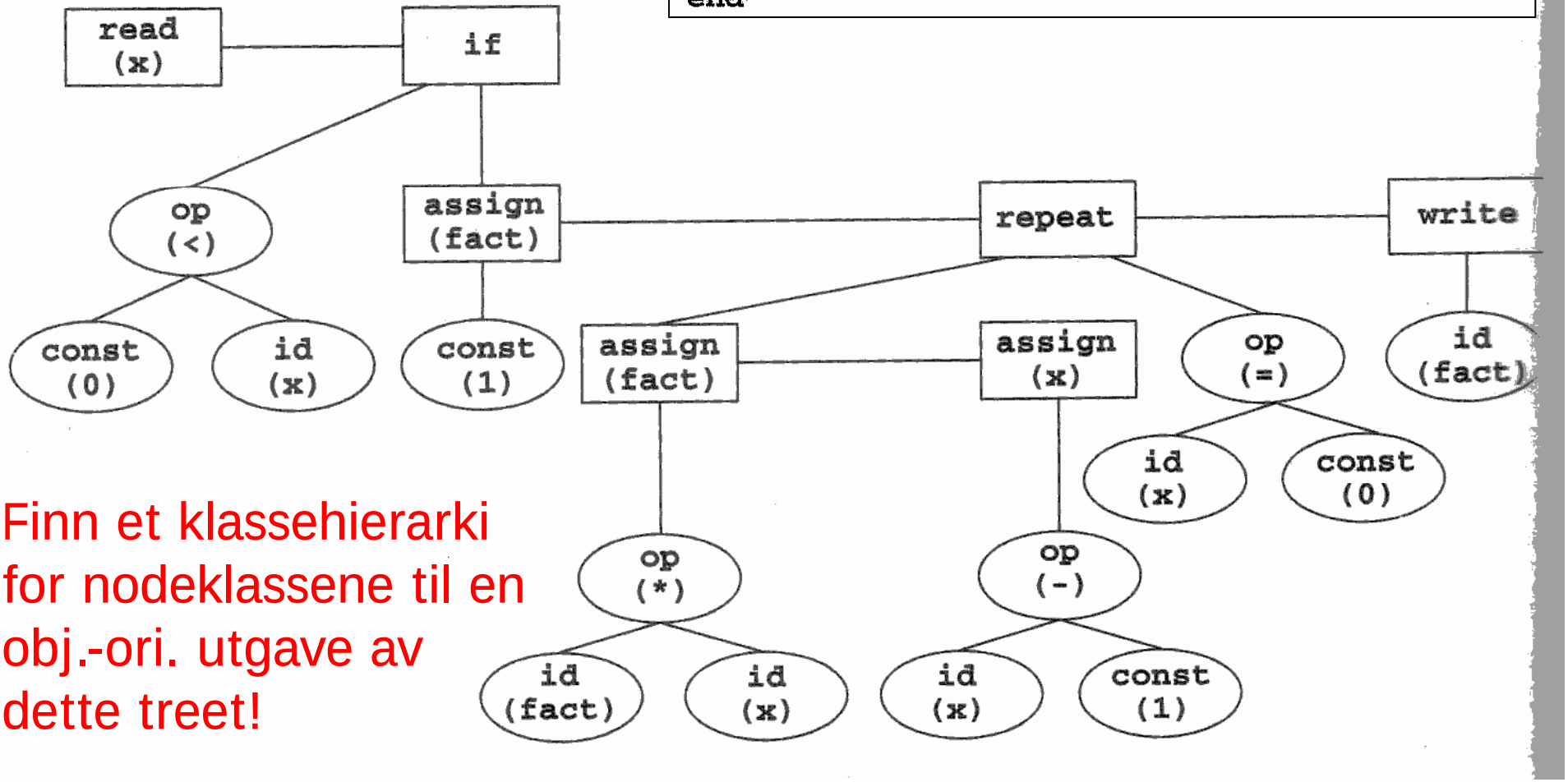
Svar på spørsmålene på forrige foil

- 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 assossiativites-angivelse, og er nok derved entydig. If-setninger med både **else** og **end** er ikke noe problem for entydigheten.
- 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

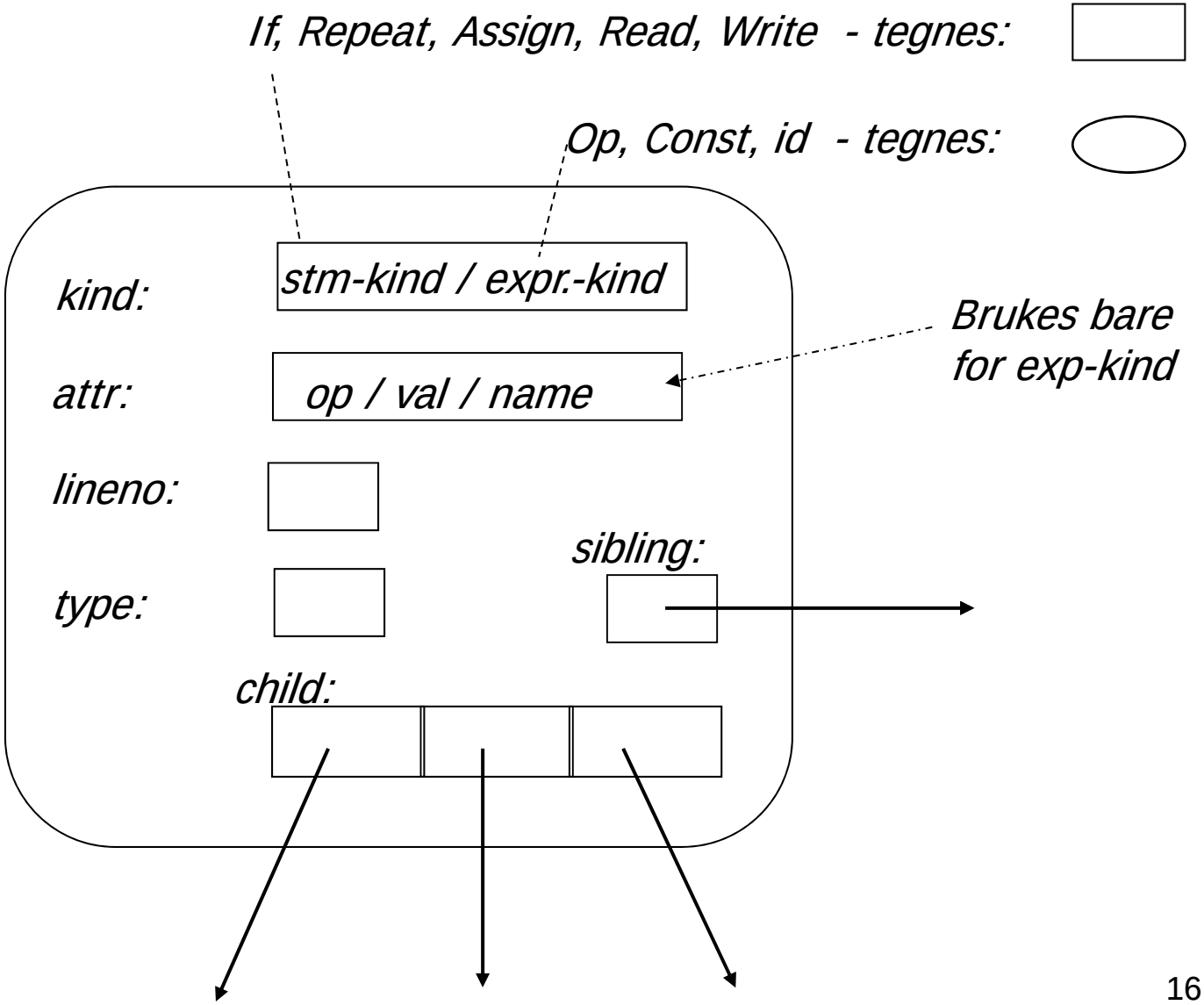
```
read x; { input an integer }
if 0 < x then { don't compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1
  until x = 0;
  write fact { output factorial of x }
end
```

Abstrakt syntakstre for Tiny-programmet:



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

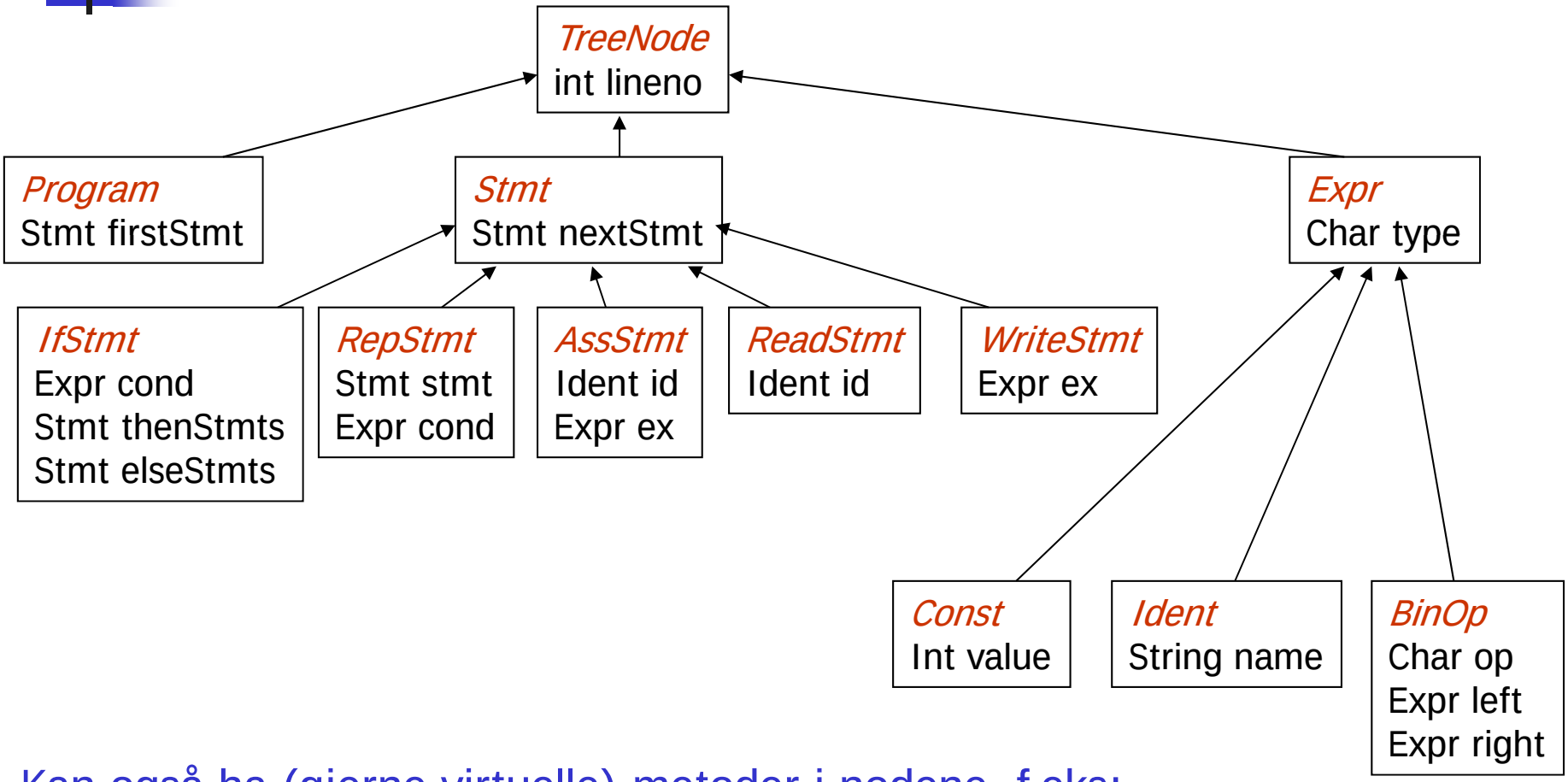
Nodestruktur i C for Tiny



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

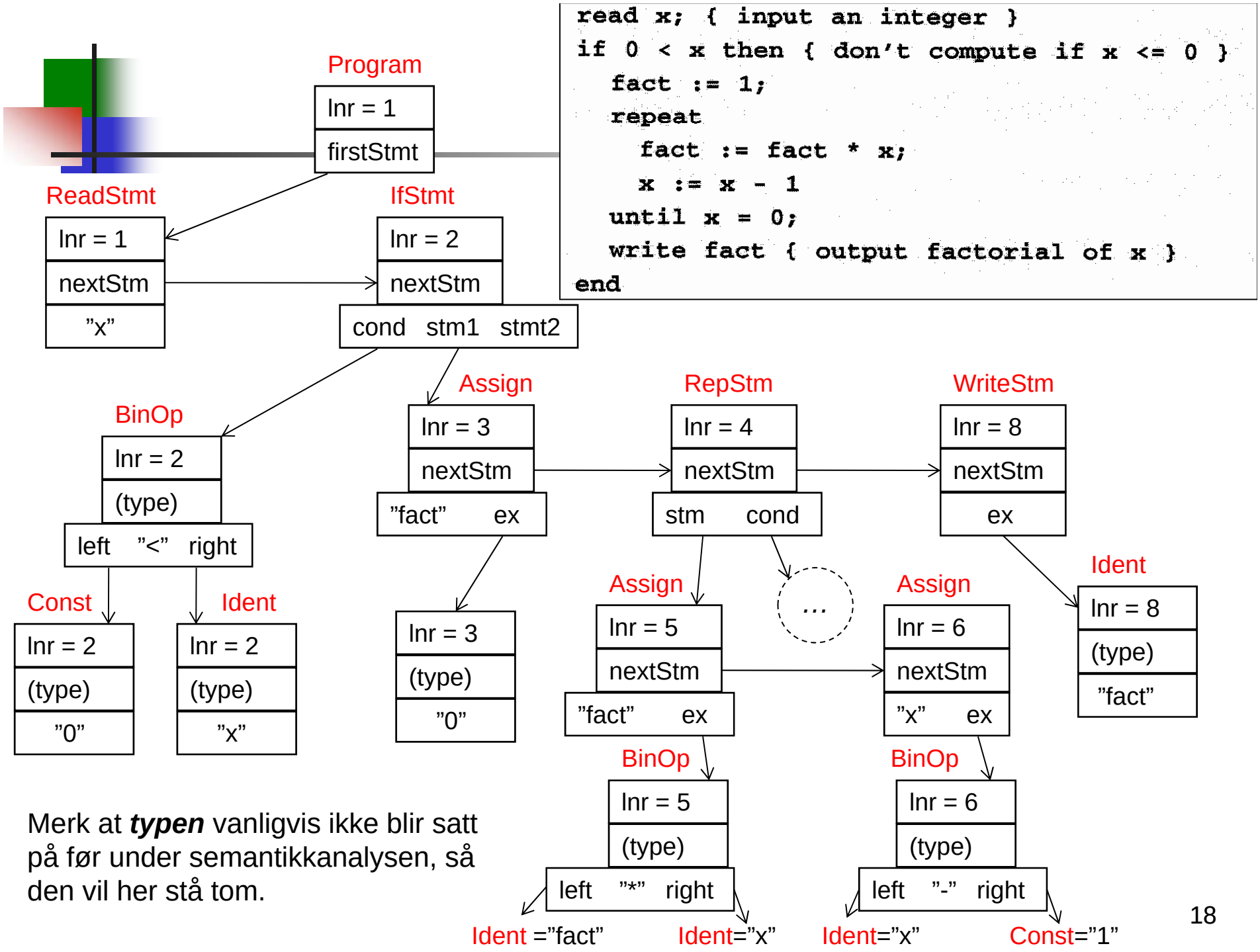
Nodeklasser for OO-utgave av abst.-synt.-tre for Tiny-språket

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:

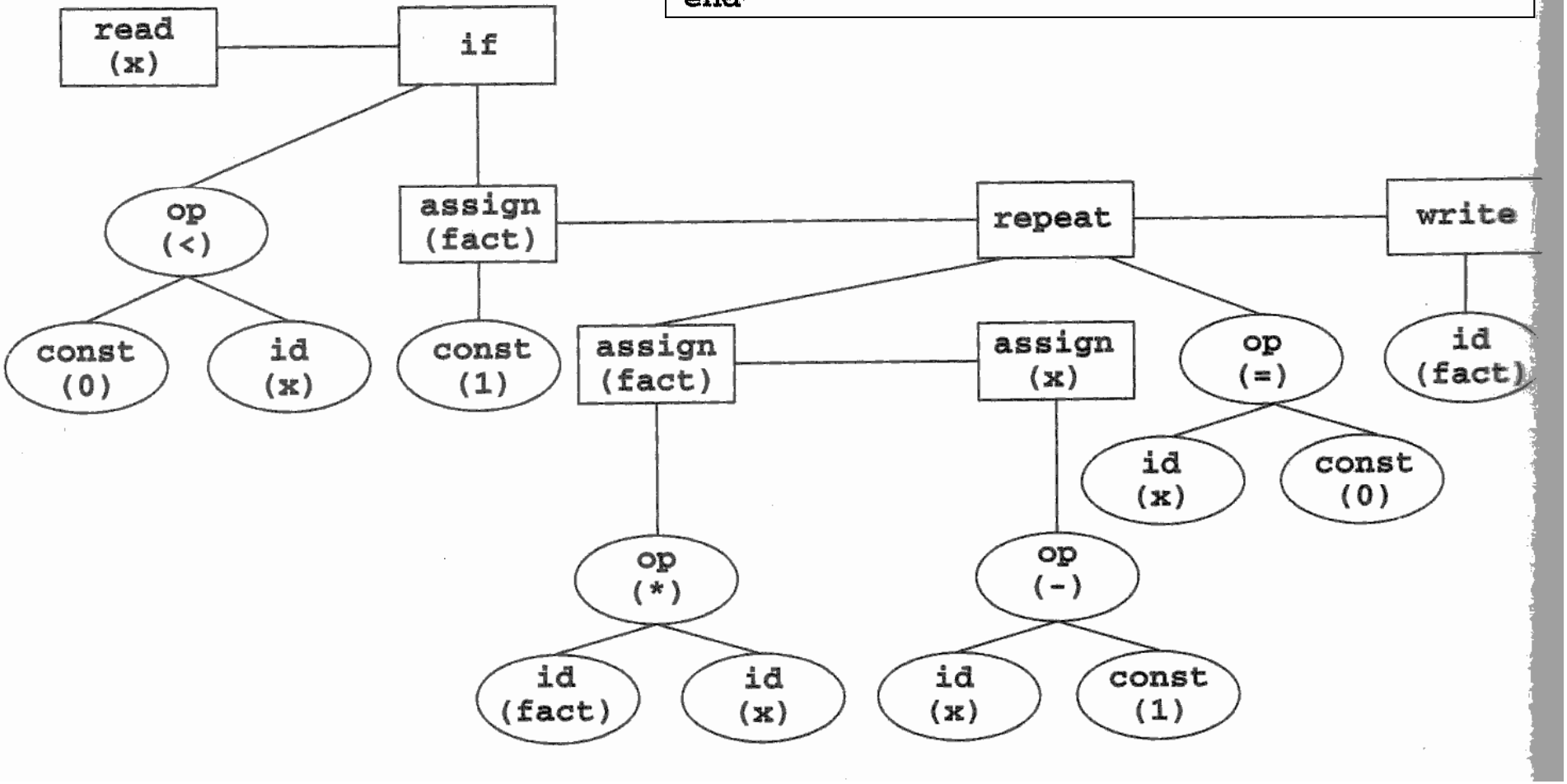
- doSemAnalyses(); Gjør semantisk analyse av noden, og av subtreet det er rot i
- generateCode(); Genererer kode for noden, og for subtreet det er rot i



Merk at ***typen*** vanligvis ikke blir satt på før under semantikkanalysen, så den vil her stå tom.

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
```



Entydig grammatikk for uttrykk med eksponensiering

Uten eksponensiering:

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

Med eksponensiering. Vi treneger en ny ikke-terminal ”expon”:

exp -> *exp addop term* | *term*

addop -> **+** | **-**

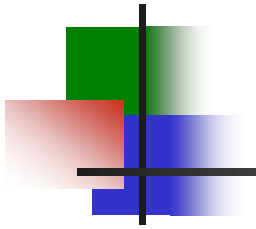
term -> *term mulop factor*

mulop -> *****

factor -> *expon exponop factor* Høyre-assosiativ!

exponop -> **^**

expon -> **(** *exp* **)** | **number**



Algoritme for å beregne mengden av utnullbare ikke-teminaler (uten å beregne First)

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

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

1. Først legges alle ikke-terminaler A som har en produksjon A -> ε, inn i UNB
2. (Gjentas til det ikke blir mer forandring):

   Om en ikke-terminal A har et alternativ:

   A -> ... | B C ... G | ...

   der alle B C ... G er ikke-terminaler som allerede er med i UNB, så legg også A inn i UNB.