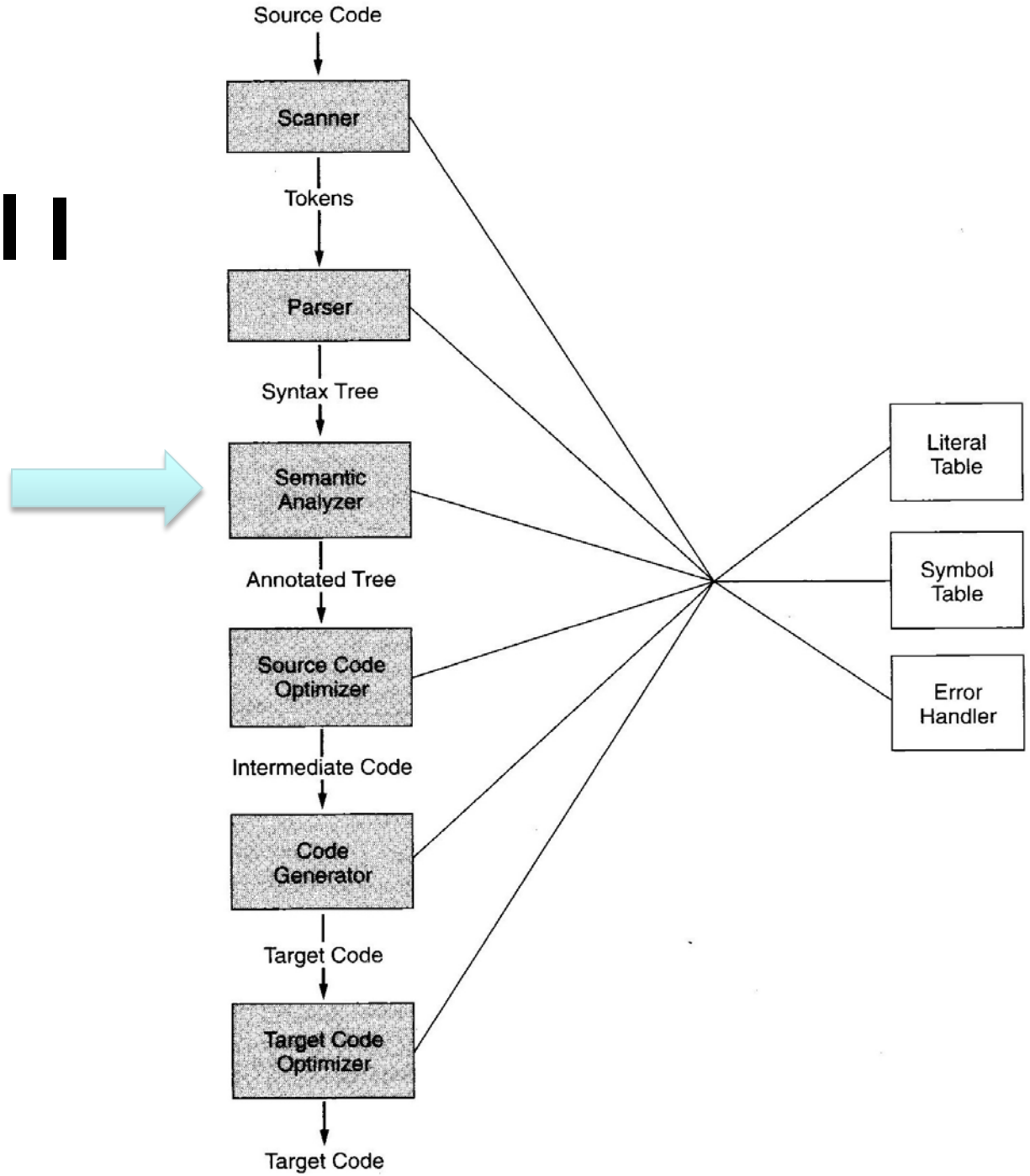
Semantisk Analyse del I

Attributtgrammatikker

Kapittel 6.1-6.2



INF 5110 - 2013

Statisk semantisk analyse – kapittel 6: Innhold

- Generelt om statisk semantisk analyse
- Attributt-grammatikker (kapittel 6.1-6.2)
- Symboltabell (kapittel 6.3)
- Datatyper og typesjekking (kapittel 6.4)

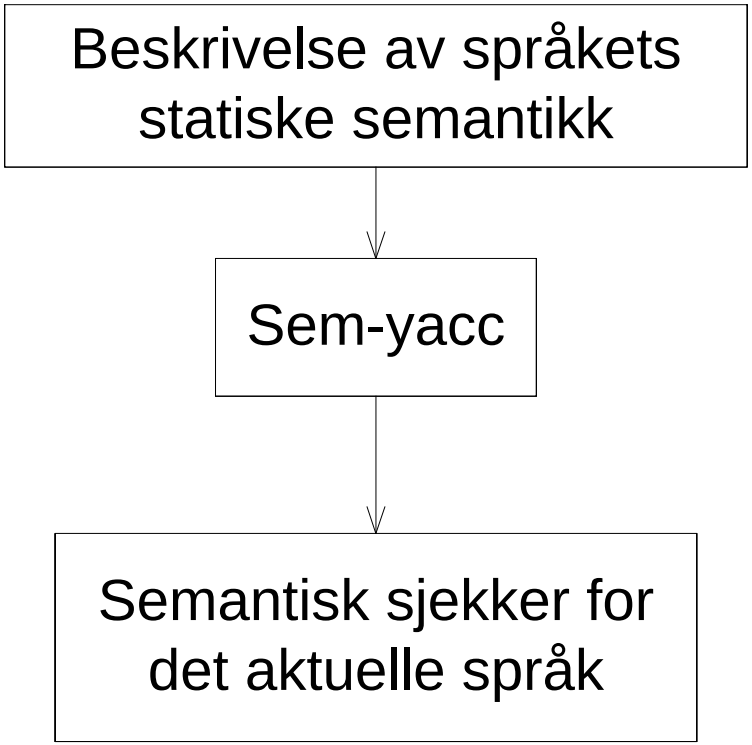
Generelt om semantisk analyse

- Formål: sjekke alle krav i språkdefinisjonen som
  - kan sjekkes før utførelsen.
  - ikke naturlig sjekkes under syntaktisk analyse.

- Typisk:
  - at bruk av navn er konsistent med deres deklarasjon.
  - at typen av (sub)-uttrykk stemmer med operasjonene.

- Men:
  - Ikke alt kan sjekkes før utførelsen.
  - Stor forskjell på språk med og uten typer på variable og parametre.

En drøm



Men:

- Mangler standard beskrivelsesspråk.
- Komplisert input til semantisk sjekker.
- Alltid mange ad-hoc regler.

Derfor:

- Ser på generelle metoder.
- Må programmeres spesielt i hvert tilfelle.

ATTRIBUTTGRAMMATIKKER

Attributter

- En attributt er en egenskap ved et språkbegrep.
- Eksempler:
  - Datatypen til en variabel.
  - Verdien til ett uttrykk.
  - Lokasjonen til en variabel i minnet.
  - Objekt-koden til en metode.
  - Antall signifikante sifre i et tall.

- Statiske attributter: Kan beregnes før utførelsen
- Dynamiske attributter: Må beregnes under utførelsen

- For attributt-grammatikker er alle attributter statiske, og de er definert i tilknytning til grammatikken for språket

Attributt-grammatikker

- Attributter er variable knyttet til nodene i parseringstreet.
- Deres verdier er definert ved semantiske regler.
- Hver semantiske regel er knyttet til en produksjon i grammatikken.
- Eksempel:

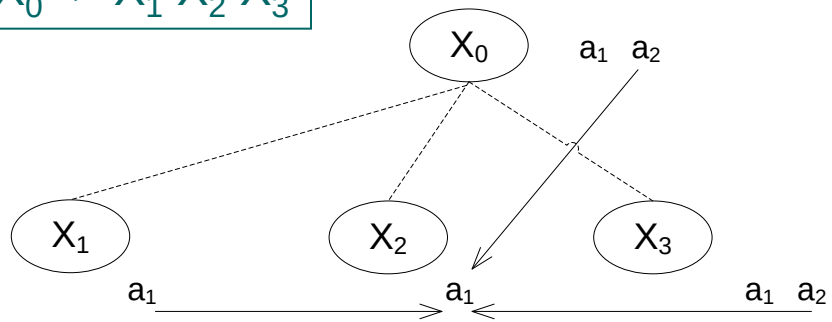
Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ **+** $term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2$ **-** $term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2$ ***** $factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow$ **(** exp **)**	$factor.val = exp.val$
$factor \rightarrow$ **number**	$factor.val =$ **number**$.val$

Attributt-grammatikker

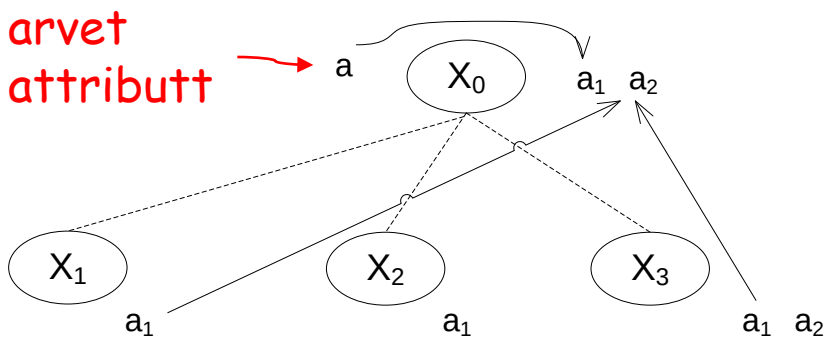
- Gitt en grammatikk på ren BNF-form
  - For hvert grammatikk-symbol X (terminal eller ikke-terminal) skal det være gitt en mengde (navnede) attributter.
  - Attributt-mengdene for de forskjellige symbolene kan generelt være helt forskjellige, men har ofte mye felles.
  - Attributtene er ment å materialisere seg som variable knyttet til nodene i et parseringstre.
  - Attributtet a til noden X skrives X.a.
  - Attributtene er definert ved at det til hver regel er knyttet en likning på formen:

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

X_0 -> X_1 X_2 X_3



arvet attributt



syntetisert attributt

Syntetiserte attributter

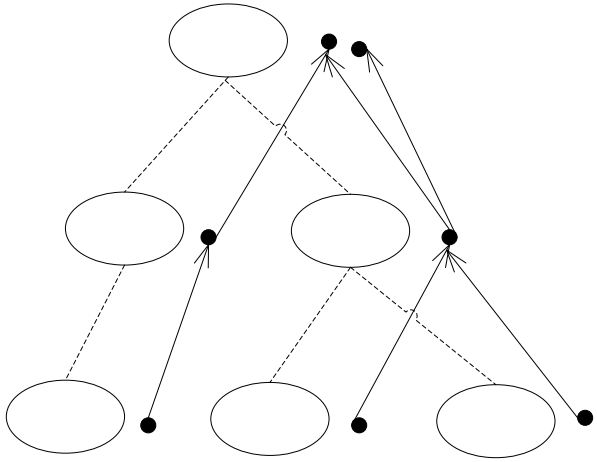
An attribute is **synthesized** if all its dependencies point from child to parent in the parse tree. Equivalently, an attribute a is synthesized if, given a grammar rule $A \rightarrow X_1 X_2 \ldots X_n$, the only associated attribute equation with an a on the left-hand side is of the form

$$A.a = f(X_1.a_1, \ldots, X_1.a_k, \ldots, X_n.a_1, \ldots, X_n.a_k)$$

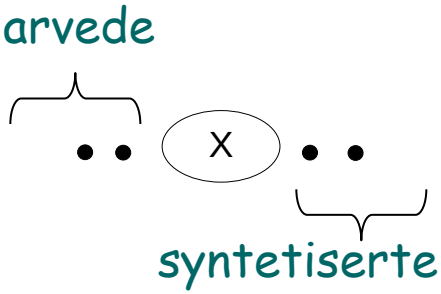
An attribute grammar in which all the attributes are synthesized is called an **S-attributed grammar**.

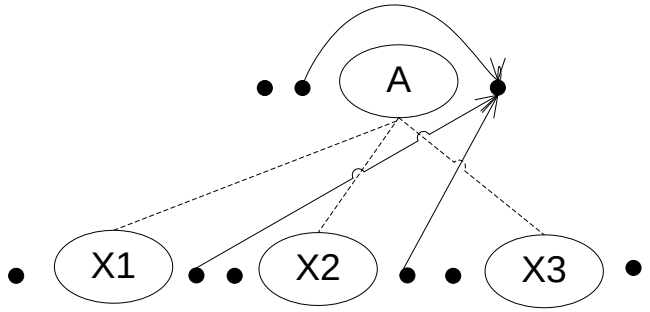
Trykkfeil: A.a kan også være avhengig av A.b dersom b er et arvet attributt.

- NB: Hvert attributt må velges til enten å være syntetisert eller arvet.



Avhengighet i en S-attributt-grammatikk





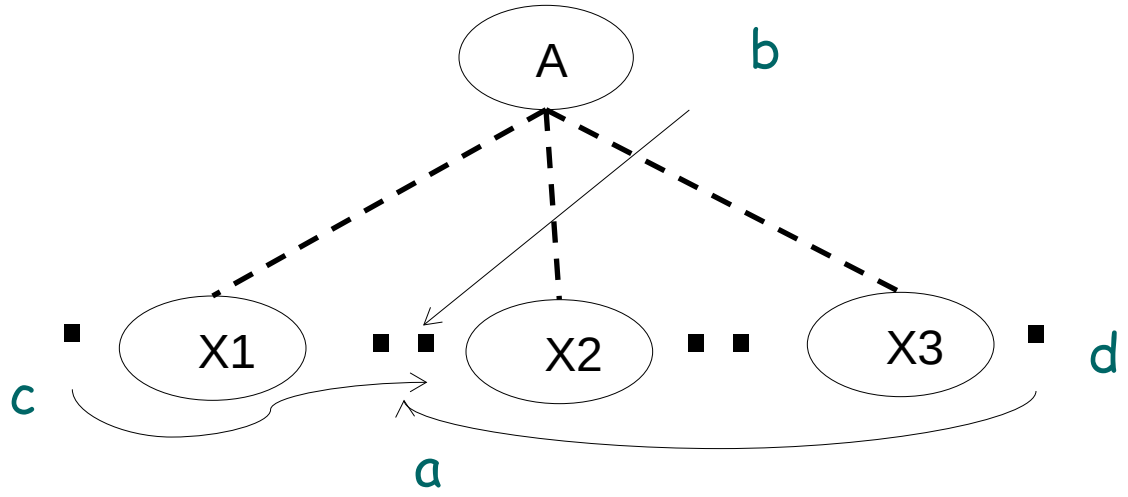
Mulig avhengighet for syntetisert attributt

Arvede attributter

- ”An attribute that is not synthesized is called an inherited attribute.”

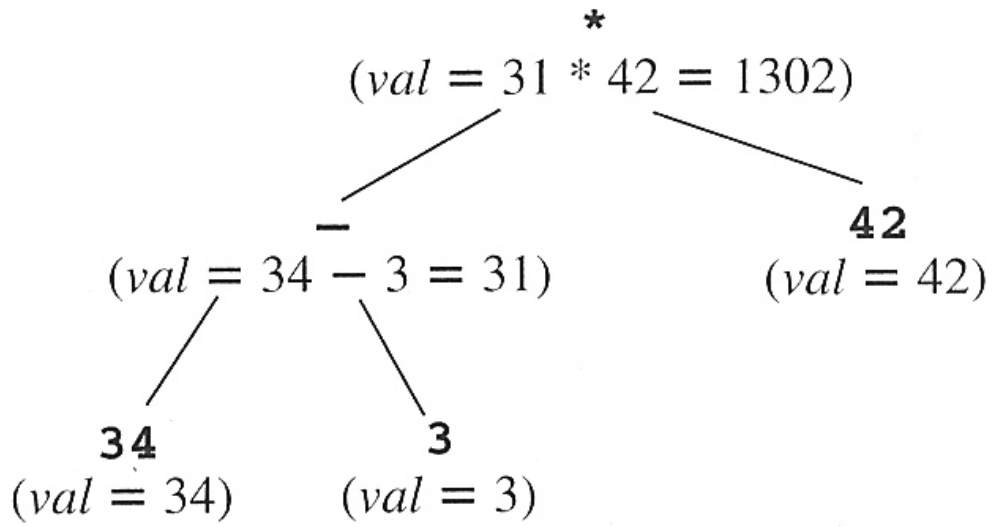
Heller:
et attributt sies å være arvet, når det defineres for et symbol på høyresiden av de produksjoner det opptrer i

$A \rightarrow X_1\ X_2\ X_3$ $\qquad$ $X_2.a = A.b + X_1.c + X_3.d$



Man kan gjerne tenke i

- abstrakte syntakstrær
- flertydige grammatikker



Attributter
exp: val
number: val (fra scanneren)
terminaler: -

$$exp \rightarrow exp \texttt{ + } exp \mid exp \texttt{ - } exp \mid exp \texttt{ * } exp \mid \texttt{(} exp \texttt{)} \mid \textbf{number}$$

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2 \texttt{ + } exp_3$	$exp_1.val = exp_2.val + exp_3.val$
$exp_1 \rightarrow exp_2 \texttt{ - } exp_3$	$exp_1.val = exp_2.val - exp_3.val$
$exp_1 \rightarrow exp_2 \texttt{ * } exp_3$	$exp_1.val = exp_2.val * exp_3.val$
$exp_1 \rightarrow \texttt{(} exp_2 \texttt{)}$	$exp_1.val = exp_2.val$
$exp \rightarrow \textbf{number}$	$exp.val = \textbf{number}.val$

Attr.-grammatikk som 'bygger' abstrakt syntakstre

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ **+** $term$	$exp_1.tree =$ $mkOpNode(+, exp_2.tree, term.tree)$
$exp_1 \rightarrow exp_2$ **-** $term$	$exp_1.tree =$ $mkOpNode(-, exp_2.tree, term.tree)$
$exp \rightarrow term$	$exp.tree = term.tree$
$term_1 \rightarrow term_2$ ***** $factor$	$term_1.tree =$ $mkOpNode(*, term_2.tree, factor.tree)$
$term \rightarrow factor$	$term.tree = factor.tree$
$factor \rightarrow$ **(** exp **)**	$factor.tree = exp.tree$
$factor \rightarrow$ **number**	$factor.tree =$ $mkNumNode($**number**$.lexval)$

Attributter:

- tree (for exp, term og factor)
- lexval (for number)

Merk: det abstrakte syntakstreet bygges ut fra at vi har det konkrete parseringstreet

Grammatikk med arvede attributter

Grammar Rule	Semantic Rules
$decl \rightarrow type\ var\text{-}list$	$var\text{-}list.dtype = type.dtype$
$type \rightarrow$ **`int`**	$type.dtype = integer$
$type \rightarrow$ **`float`**	$type.dtype = real$
$var\text{-}list_1 \rightarrow$ **`id,`** $var\text{-}list_2$	**`id`** $.dtype = var\text{-}list_1.dtype$
	$var\text{-}list_2.dtype = var\text{-}list_1.dtype$
$var\text{-}list \rightarrow$ **`id`**	**`id`** $.dtype = var\text{-}list.dtype$

Attributter:

- dtype (for type, varlist, id)

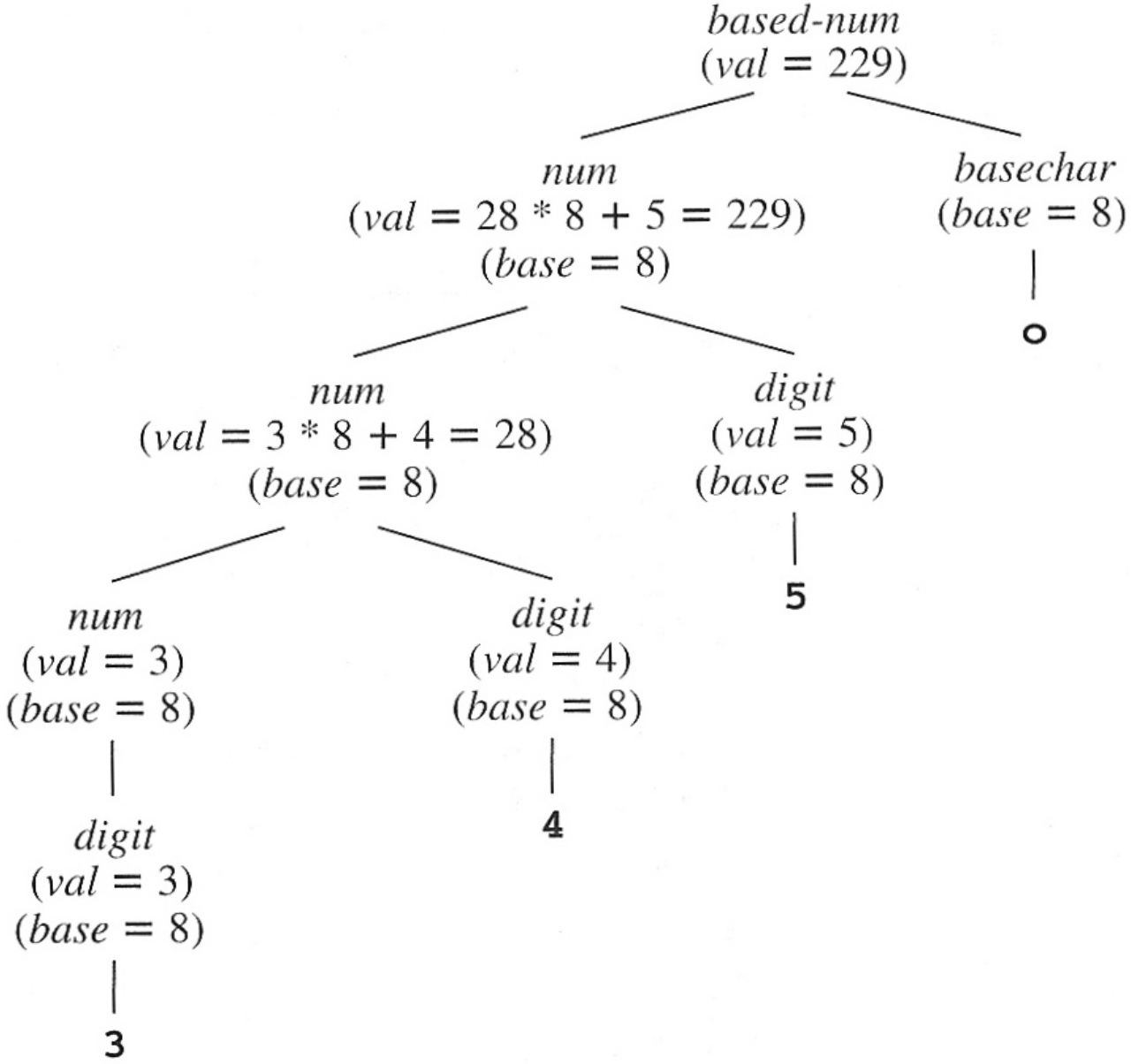
Eksempel med arvede og syntetiserte attributter

Grammar Rule	Semantic Rules
based-num → *num basechar*	*based-num.val* = *num.val* *num.base* = *basechar.base*
basechar → **o**	*basechar.base* = 8
basechar → **d**	*basechar.base* = 10
$num_1 \rightarrow num_2\ digit$	$num_1.val$ = **if** *digit.val* = *error* **or** $num_2.val$ = *error* **then** *error* **else** $num_2.val * num_1.base + digit.val$ $num_2.base = num_1.base$ $digit.base = num_1.base$
num → *digit*	*num.val* = *digit.val* *digit.base* = *num.base*
digit → **0**	*digit.val* = 0
digit → **1**	*digit.val* = 1
. . .	. . .
digit → **7**	*digit.val* = 7
digit → **8**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 8
digit → **9**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 9

Attributter:

- val
  (for based-num, num og digit)
- base
  (for basechar og num)

Eksempel based-num forts.



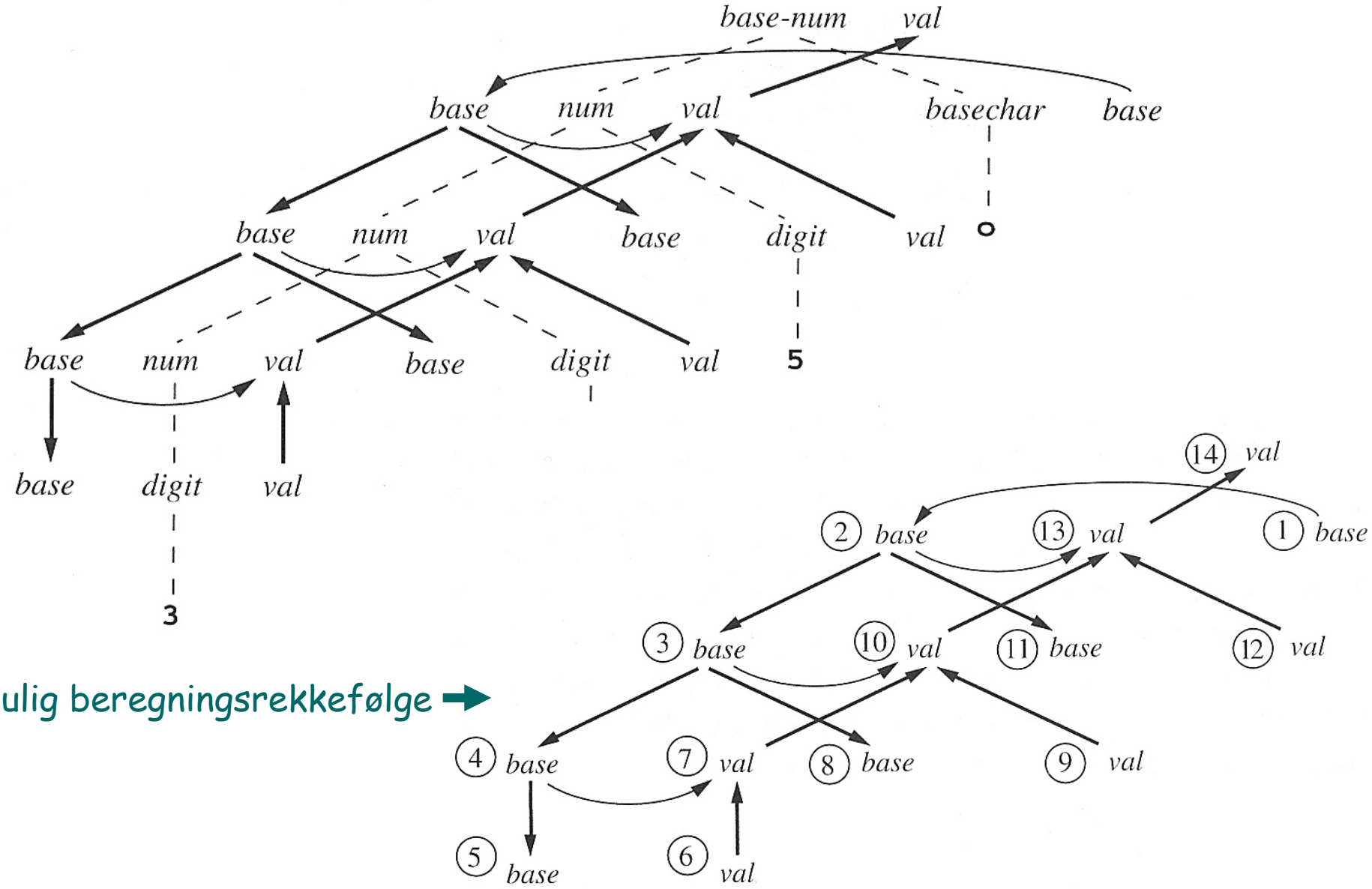
Konsistens og beregnbarhet

- Alle attributter er enten arvede eller syntetiserte
  - (Men OK at basechar.base er syntetisert, mens num.base er arvet siden dette er to forskjellige attributter.)

- Alle attributter må være definerte. Gjøres ved at innen hver produksjon må:
  - alle syntetiserte attributter i venstresiden være definert.
  - alle arvede attributter i høyresiden være definert.
  - det ikke være noen lokale avhengighets-løkker.

- Siden alle attributter er enten arvede eller syntetiserte, er hvert attributt i et parseringstre definert en og bare en gang.

Avhengighetsgraf for based-num - eksempelet



Mulig beregningsrekkefølge ➡

INF 5110 - 2013

Omskriving fra arvede til syntetiserte attributter

- Denne ga arvede attributter:

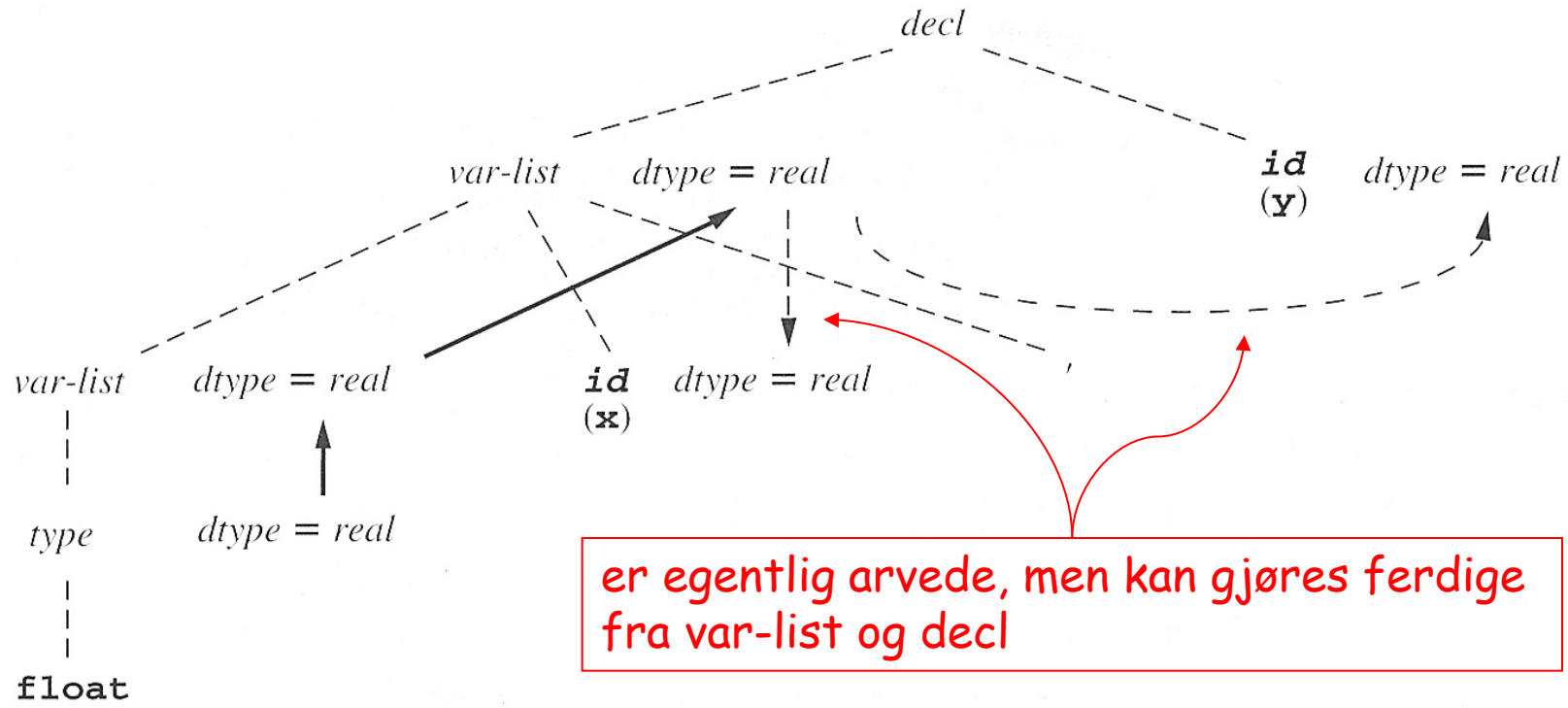
$$
\begin{aligned}
&\textit{decl} \rightarrow \textit{type var-list}\\
&\textit{type} \rightarrow \texttt{int} \mid \texttt{float}\\
&\textit{var-list} \rightarrow \texttt{id ,}\ \textit{var-list} \mid \texttt{id}
\end{aligned}
$$

- Nytt forslag:

$$
\begin{aligned}
&\textit{decl} \rightarrow \textit{var-list}\ \texttt{id}\\
&\textit{var-list} \rightarrow \textit{var-list}\ \texttt{id ,} \mid \textit{type}\\
&\textit{type} \rightarrow \texttt{int} \mid \texttt{float}
\end{aligned}
$$

Med ny grammatikk:

Grammar Rule	Semantic Rules
decl → *var-list* **id**	**id** .*dtype* = *var-list.dtype*
$var\text{-}list_1 \rightarrow var\text{-}list_2$ **id** ,	$var\text{-}list_1$.*dtype* = $var\text{-}list_2$.*dtype*
	id .*dtype* = $var\text{-}list_1$.*dtype*
var-list → *type*	*var-list.dtype* = *type.dtype*
type → **int**	*type.dtype* = *integer*
type → **float**	*type.dtype* = *real*



er egentlig arvede, men kan gjøres ferdige fra var-list og decl

Løkker i avhengighets-grafen

- Ikke i noe syntakstre må det være løkker i avhengighetsgrafen!
- Generelt er dette vanskelig å garantere.
- Man kan sjekke det **etter** at treet er bygget. Om ikke løkker: Finn ’topologisk sortering’ som gir brukbar rekkefølge.
- I praksis:
  - Man overbeviser seg på forhånd om at det ikke kan bli løkker i noe syntakstre.
  - Forhåndsbestemmer en rekkefølge for beregning.

BEREGNINGSALGORITMER

Eksempel: type-grammatikk

Grammar Rule	Semantic Rules
decl → *type var-list*	*var-list.dtype* = *type.dtype*
type → **`int`**	*type.dtype* = *integer*
type → **`float`**	*type.dtype* = *real*
$var\text{-}list_1$ → **`id,`** $var\text{-}list_2$	**`id`** *.dtype* = $var\text{-}list_1$*.dtype*
	$var\text{-}list_2$*.dtype* = $var\text{-}list_1$*.dtype*
var-list → **`id`**	**`id`** *.dtype* = *var-list.dtype*

```
procedure EvalType ( T: treenode );
begin
   case nodekind of T of
   decl:
        EvalType ( type child of T );
        Assign dtype of type child of T to var-list child of T;
        EvalType ( var-list child of T );
   type:
        if child of T = int then T.dtype := integer
        else T.dtype := real;
   var-list:
        assign T.dtype to first child of T;
        if third child of T is not nil then
            assign T.dtype to third child;
            EvalType ( third child of T );
   end case;
end EvalType;
```

Eksempel: beregning for base-grammatikk

based-num → *num basechar*
basechar → `o` | `d`
num → *num digit* | *digit*
digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

```
procedure EvalWithBase ( T: treenode );
begin
  case nodekind of T of
  based-num:
      EvalWithBase ( right child of T );
      assign base of right child of T to base of left child;
      EvalWithBase ( left child of T );
      assign val of left child of T to T.val;
  num:
      assign T.base to base of left child of T;
      EvalWithBase ( left child of T );
      if right child of T is not nil then
        assign T.base to base of right child of T;
        EvalWithBase ( right child of T );
        if vals of left and right children ≠ error then
            T.val := T.base*(val of left child) + val of right child
        else T.val := error;
      else T.val := val of left child;
  basechar:
      if child of T = o then T.base := 8
      else T.base := 10;
  digit:
      if T.base = 8 and child of T = 8 or 9 then T.val := error
      else T.val := numval ( child of T );
  end case;
end EvalWithBase;
```

Alternativ til EvalWithBase

For 'basechar' er den 'base', ellers er den 'val'

Mangler teknisk en parameter, men uten betydning

```
function EvalWithBase ( T: treenode; base: integer ): integer;
var temp, temp2: integer;
begin
   case nodekind of T of
   based-num:
      temp := EvalWithBase ( right child of T );
      return EvalWithBase ( left child of T, temp );
   num:
      temp := EvalWithBase ( left child of T, base );
      if right child of T is not nil then
         temp2 := EvalWithBase ( right child of T, base );
         if temp ≠ error and temp2 ≠ error then
               return base*temp + temp2
         else return error;
      else return temp;
   basechar:
      if child of T = o then return 8
      else return 10;
   digit:
      if base = 8 and child of T = 8 or 9 then return error
      else return numval ( child of T );
   end case;
end EvalWithBase;
```

- Ny filosofi
  - Returverdier er syntetiserte attributter
  - Parametere er arvede attributter

For beregning under parsering venstre –> høyre

An attribute grammar for attributes $a_1, \ldots, a_k$ is **L-attributed** if, for each inherited attribute a_j and each grammar rule

$$X_0 \rightarrow X_1 X_2 \ldots X_n$$

the associated equations for a_j are all of the form

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots, X_{i-1}.a_1, \ldots, X_{i-1}.a_k)$$

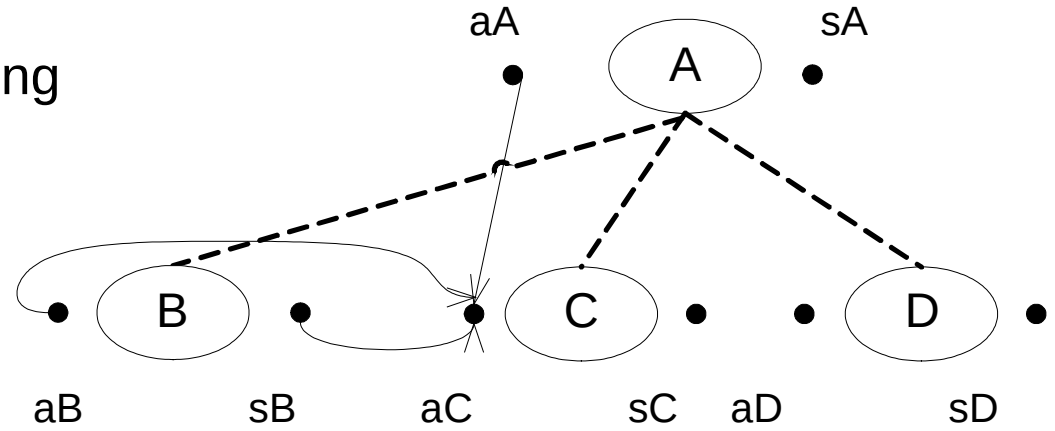
That is, the value of a_j at X_i can only depend on attributes of the symbols $X_0, \ldots, X_{i-1}$ that occur to the left of X_i in the grammar rule.

Trykkfeil: Krever også $X_0.a_1$, ... $X_o.a_k$ må alle være arvede attributter (ellers kan det oppstå løkker, og man kan få høyre->venstre avhengigheter

Beregning av L-attribut-gramm.

- under recursive descent parsering

A -> ... | B C D | ...




```
int A(aA){
  ...
  if <A-> BCD skal velges> then {
    int aB,sB,aC,sC,aD,sD
    aB = fB(aA);
    sB = B(aB);
    aC = fC(aA,aB,sB);
    sC = C(aC);
    aD = fD(aA,aB,sB,aC,sC);
    sD = D(aD);
    return gA(aA,aB,sB,aC,sC,aD,sD);
  } else
  ...
}
```

Beregning av *sA*

Neste forelesning: Fredag 1. mars

OPPGAVER: 6.5, 6.7, 6.13 + DE NESTE SIDENE

class → **class name** *superclass* { *decls* }
decls → *decls* ; *decl* | *decl*
decl → *variable-decl*
decl → *method-decl*
method-decl → *type* **name** (*params*) *body*
type → **int** |**bool** | **void**
superclass → **name**

Ord i *kursiv* er non-terminaler, ord og tegn i **fet skrift** er terminal-symboler, mens **name** representerer et navn som scanneren leverer. Det kan antas at **name** har attributtet 'name'.

Metoder med samme navn som klassen er 'konstruktører', og det gjelder følgende regel: Konstruktører må være spesifisert med typen **void**.

Lag semantiske regler for denne regel i følgende fragment av en attributtgrammatikk.

Grammar Rule	Semantic Rule
class → ***class name*** { *decls* }	
decls → *decls* ; *decl*	
decls → *decl*	
decl → *variable-decl*	*Skal ikke fylles ut*
decl → *method-decl*	
method-decl → *type* **name** (*params*) *body*	
type → **int**	
type → **bool**	
type → **void**	

Eksamen 2007 – oppgave 2b

decls → *decls* **;** *decl* | *decl*
decl → *var-decl* | *function-decl*
var-decl → *type* ***id*** **=** *expression*
function-decl → *type* ***id*** **(** *parameter*? **)** *body*
type → **int** | **bool** | **void**
parameter → *type* ***id*** | *type* **func** ***id***
call → ***id*** (***id?***)

Ord i *kursiv* er ikke-terminaler, ord og tegn i **fet** skrift er terminal-symboler. ***id*** representerer et navn.

En parameter er enten en verdi overført 'by value' eller en funksjon uten parameter. Den enkle reglen i dette språket er at en funksjon med en 'by value'-parameter bare kan kalles med en variabel som aktuell parameter (altså ikke med et generelt uttrykk), mens en funksjon med en funksjonsparameter bare kan kalles med en aktuell parameter som er en funksjon uten parametere. Typen til den aktuelle parameteren skal i begge tilfelle være samme type som den formelle. En funksjon uten parameter må kalles uten aktuell parameter.

Fyll ut de tomme felter i følgende attributtgrammatikk slik at attributtet *ok* for *call* er *true* hvis kallet er gjort ifølge disse regler, ellers *false*.

Du kan anta at det finnes semantiske regler som legger navn inn i symboltabellen. Du kan også anta at *lookup(**id**.name).kind* gir verdien 'var' for en variabel og 'func' for en funksjon, *lookup(**id**.name).type* er typen til det som ***id**.name* er navnet på (funksjon, variabel eller parameter) og at lookup(***id**.name).*has_parameter gir verdien '*yes*' eller '*no*' for en funksjon (med navnet ***id**.name*) avhengig av om funksjonen har en parameter eller ikke.

Det er ikke behov for å sjekke om funksjonsnavnet (***id***) i en *call*-setning faktisk er deklarert (du kan altså anta at det allerede er gjort ved andre mekanismer).

Grammar Rule	Semantic Rule
function-decl → **type id ( )** *body*	*function-decl.has_parameter = no*
function-decl → **type id (** *parameter* **)** *body*	*function-decl.has_parameter = yes* *function-decl.param-kind = parameter.kind* *function-decl.param-type = parameter.type*
parameter → **type id**	
parameter → **type func id**	
type → **int**	*type.type = integer*
type → **bool**	*type.type = boolean*
type → **void**	*type.type = void*
call → **id** *()*	*call.ok = (lookup(**id**.name).has_parameter=no)*
call → $\mathbf{id}_1(\mathbf{id}_2)$	*call.ok =*

Eksamen 2010 – oppgave 3a

Det følgende er en del av grammatikken for et språk med funksjoner.

func → *type* **func** *id signature stmt-list*
type → **int**
type → **bool**
stmt-list → *stmt-list stmt*
stmt-list → *stmt*
stmt → *assign-stmt*
stmt → *if-stmt*
stmt → *return-stmt*
return-stmt → **return** *exp*
exp → *id*
exp → *id* **+** *id*
exp → **true**
exp → **false**

Fyll ut de tomme felter (markert med *) i attributtgrammatikken for de relevante deler av grammatikken på neste side slik at attributtet *ok* for *return-stmt* er true hvis typen til returuttryket *exp* er det samme som typen til funksjonen, ellers false.

Du kan anta at lookup-kind(*id.name)* leverer den *type*, som er lagt inn i symboltabellen ved deklarasjonen av variablen med navnet *id.name*.

Grammar Rule	Semantic Rule	
func → **type func** *id signature* *stmt-list*		*
type → **int**	type.type = Integer	
type → **bool**	type.type = Boolean	
$stmt\text{-}list_1$ → $stmt\text{-}list_2$ *stmt*		*
stmt-list → *stmt*		*
stmt → *return-stmt*		*
return-stmt → **return** *exp*	return-stmt.ok =	
exp → *id*	exp.type = lookup(id.name)	
exp → id_1 + id_2		*
exp → **true**	exp.type = Boolean	
exp → **false**	exp.type = Boolean	