6.5 Rewrite the attribute grammar of Table 6.2 to compute a *postfix* string attribute instead of *val*, containing the postfix form for the simple integer expression. For example, the *postfix* attribute for **(34-3)*42** is "34 3 − 42 + *." You may assume a concatenation operator || and the existence of a ***number***.*strval* attribute.

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 → **(** *exp* **)**	$factor.val = exp.val$
factor → ***number***	$factor.val =$ ***number***.*val*

Oppgave 6.5

Grammar Rule	Semantic Rule
$exp_1 \rightarrow exp_2$ + term	exp_1.postfix = exp_2.postfix || term.postfix || +
$exp_1 \rightarrow exp_2$ - term	exp_1.postfix = exp_2.postfix || term.postfix || -
exp → term	exp.postfix = term.postfix
$term_1 \rightarrow term_2$ * factor	$term_1$.postfix = $term_2$. postfix || factor.postfix ||*
term → factor	term.postfix = factor.postfix
factor → (exp)	factor.postfix = exp.postfix
factor → **number**	factor.postfix = **number**.strval

INF 5110 - 2012

6.7 Consider the following grammar for simple Pascal-style declarations:

$$\begin{aligned} \textit{decl} &\rightarrow \textit{var-list} \ \texttt{:} \ \textit{type} \\ \textit{var-list} &\rightarrow \textit{var-list} \ \texttt{,} \ \texttt{id} \mid \texttt{id} \\ \textit{type} &\rightarrow \texttt{integer} \mid \texttt{real} \end{aligned}$$

Write an attribute grammar for the type of a variable.

Oppgave 6.7

decl → *var-list* **:** *type*
var-list → *var-list* **,** **id** | **id**
type → **int** | **real**

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

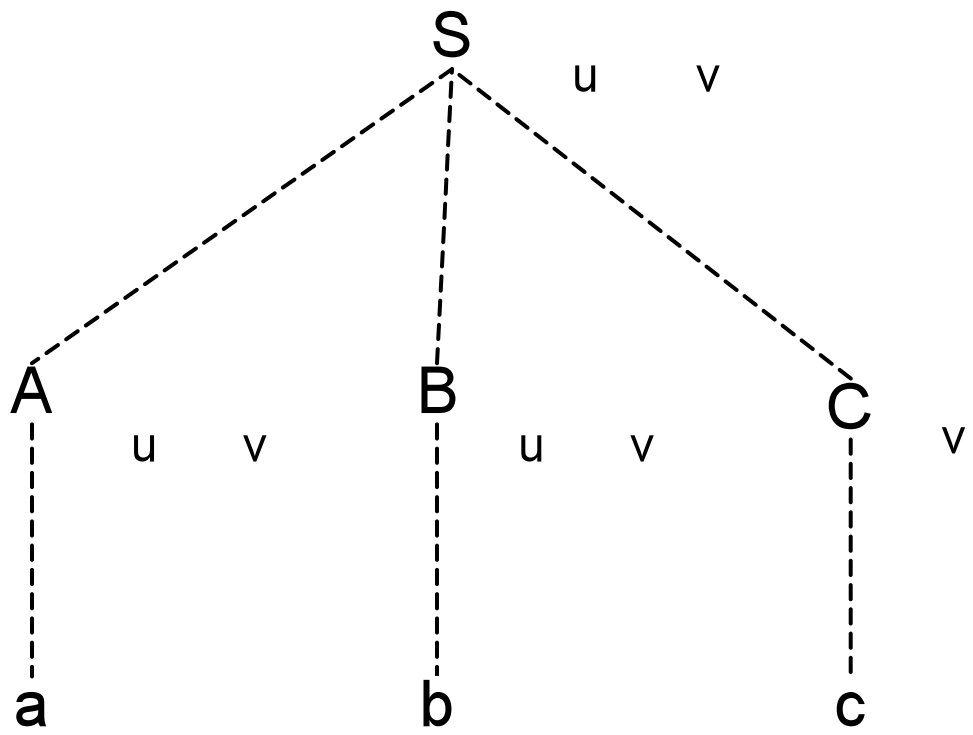
6.13 Consider the following attribute grammar:

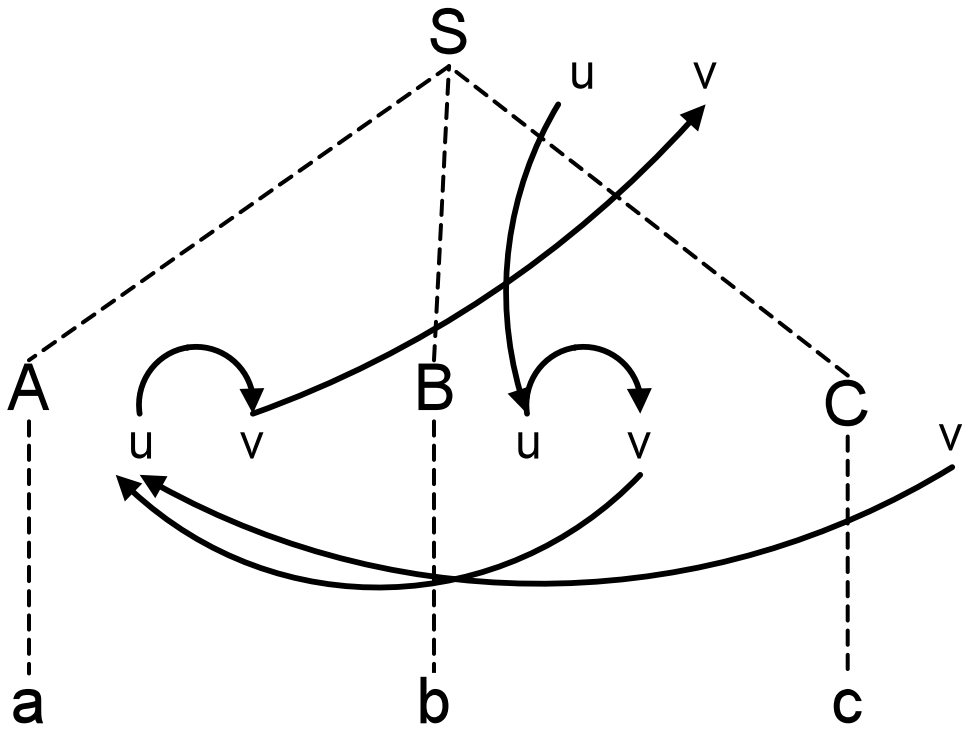
Grammar Rule	Semantic Rules
$S \rightarrow A\ B\ C$	$B.u = S.u$
	$A.u = B.v + C.v$
	$S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = 1$

a. Draw the parse tree for the string *abc* (the only string in the language), and draw the dependency graph for the associated attributes. Describe a correct order for the evaluation of the attributes.

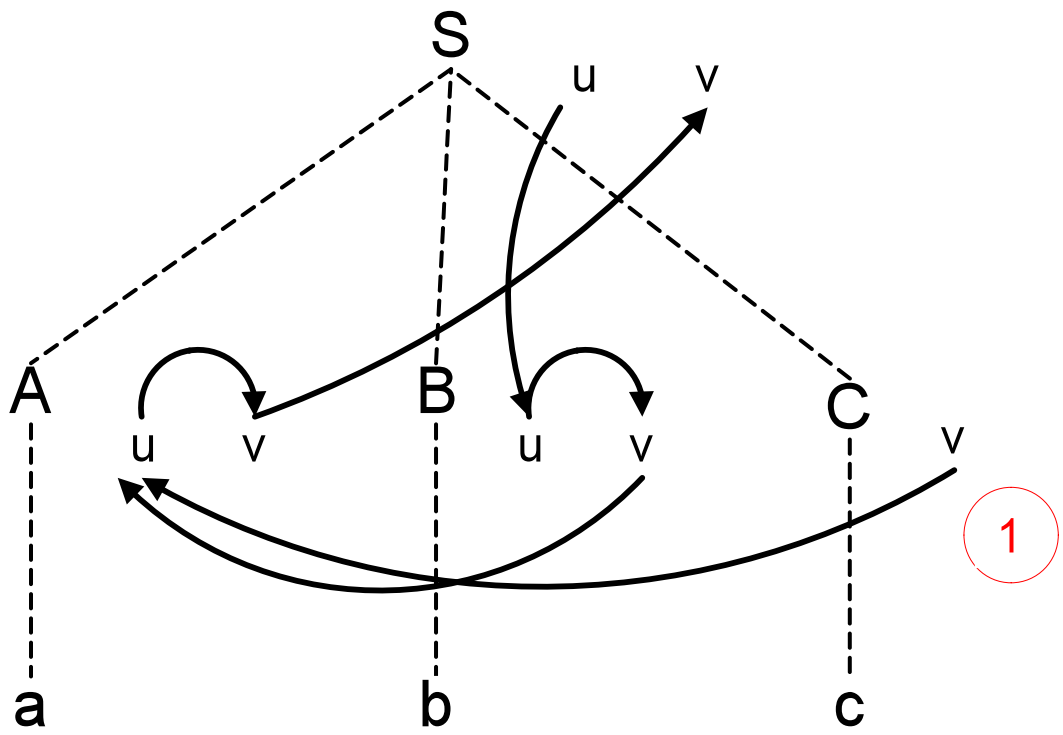
b. Suppose that $S.u$ is assigned the value 3 before attribute evaluation begins. What is the value of $S.v$ when evaluation has finished?

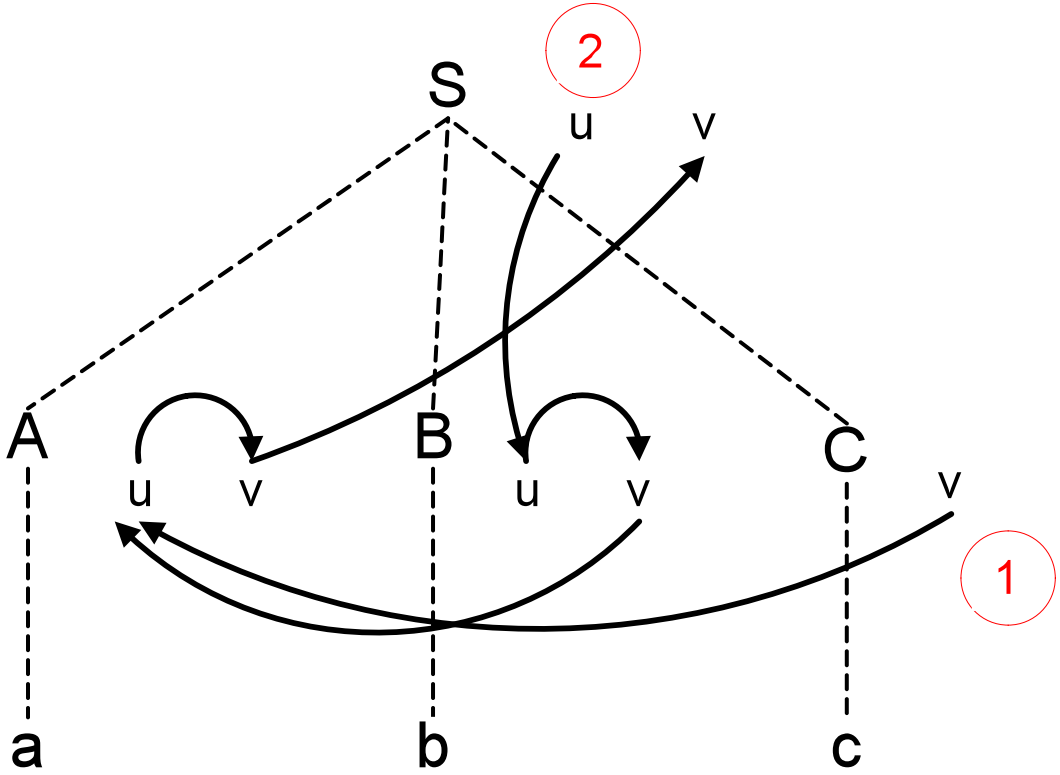
Oppgave 6.13 a 1)

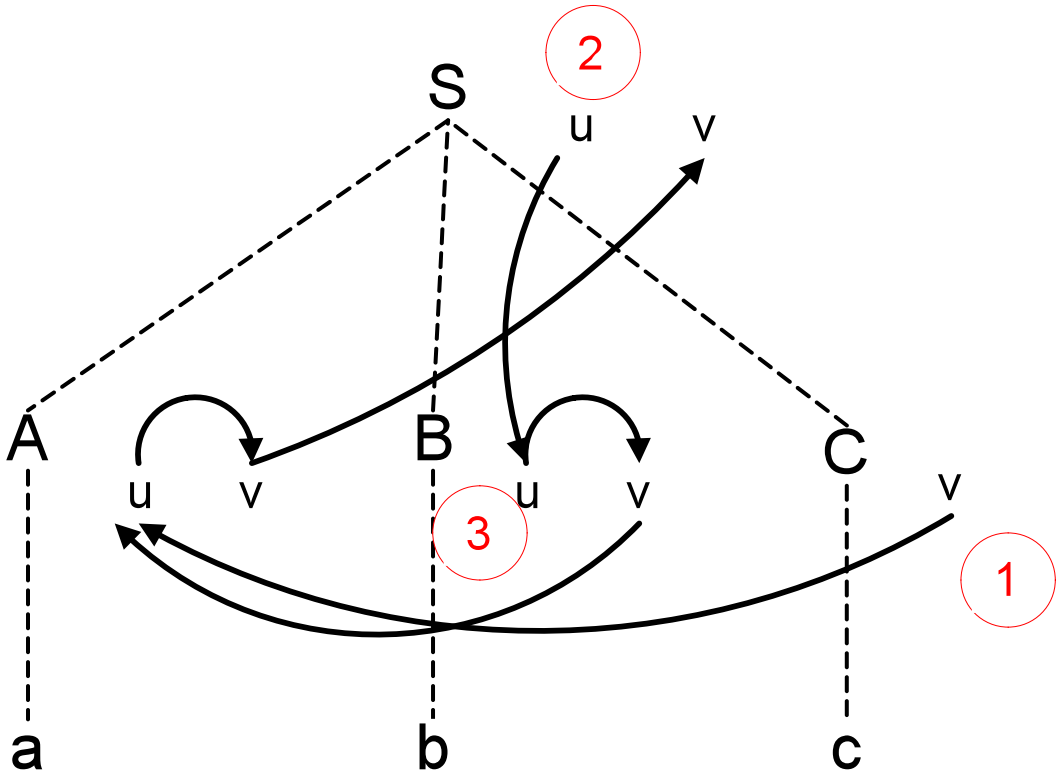


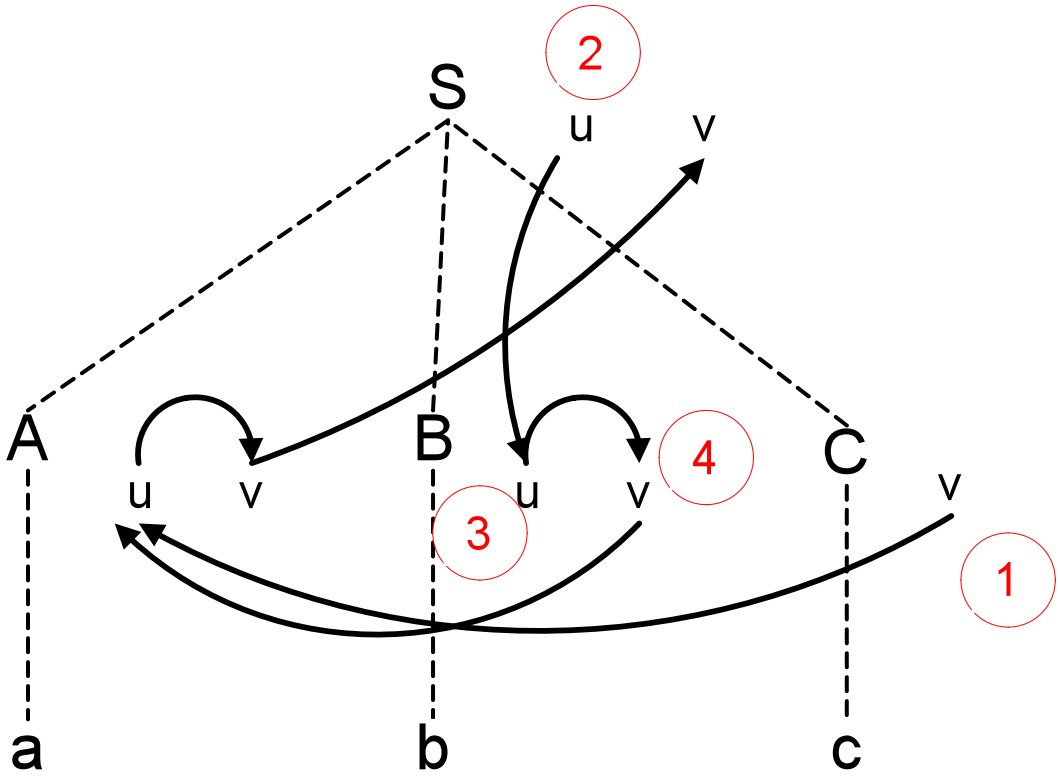

S
u
v
A
B
C
u
v
u
v
v
a
b
c

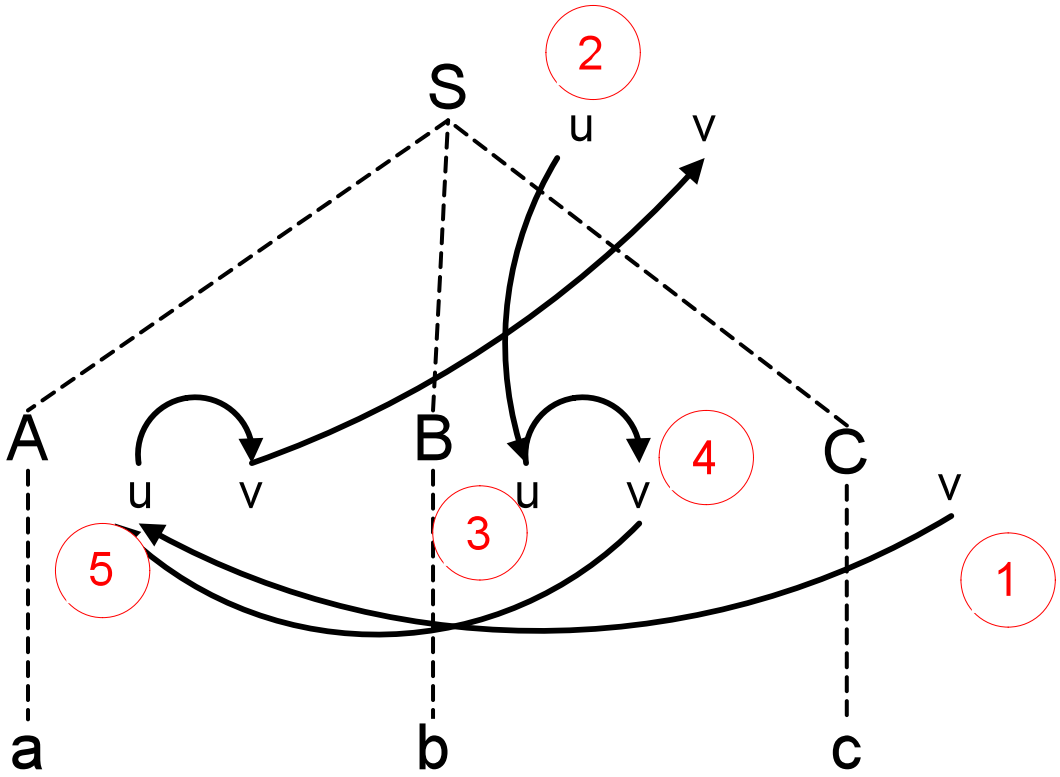
Oppgave 6.13 a 2)

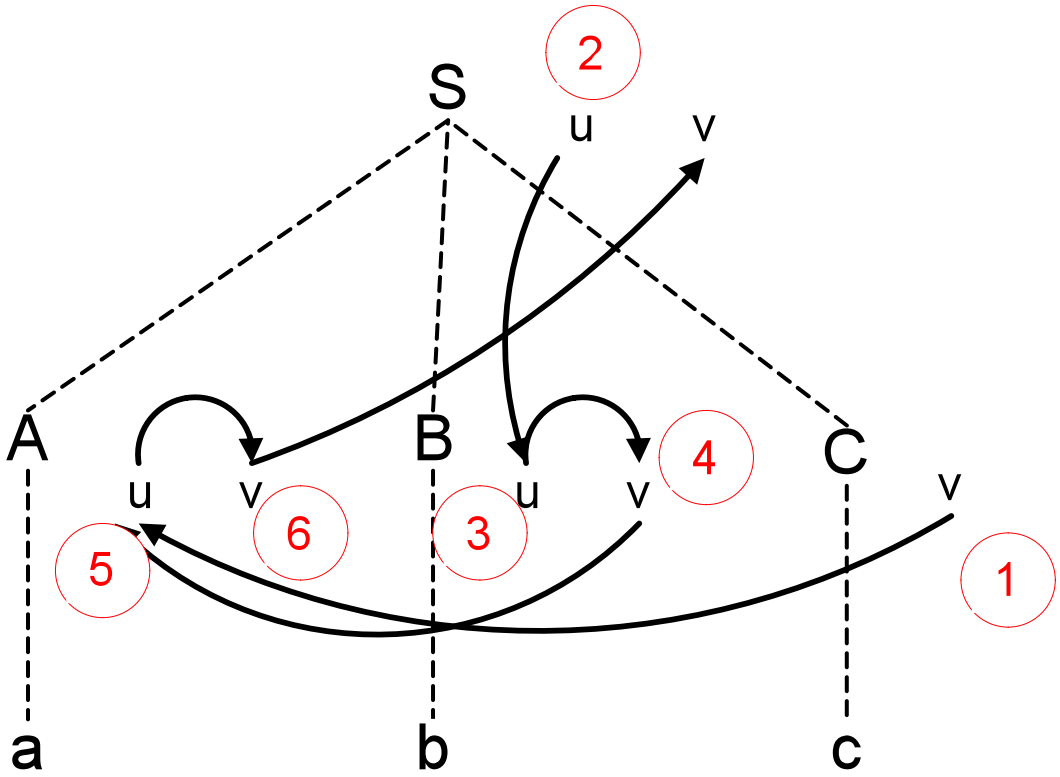


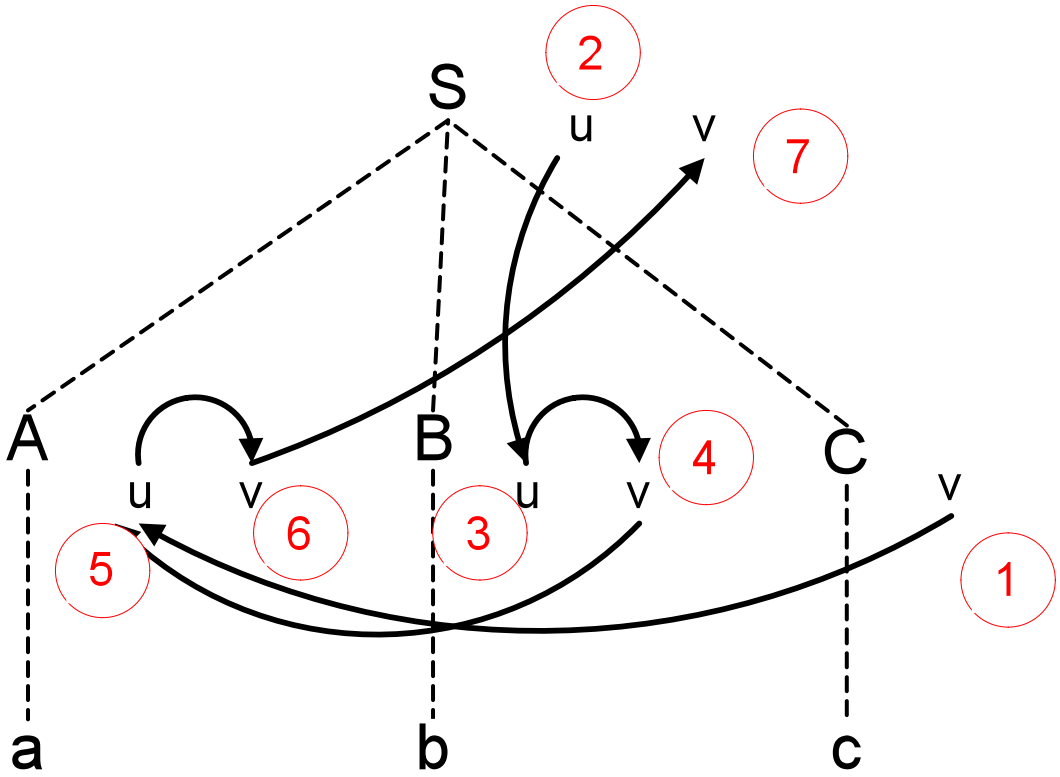

2
S
u
v
A
B
C
u
v
u
v
v
1
a
b
c


S
2
u
v
A
B
C
u
v
u
v
v
3
1
a
b
c

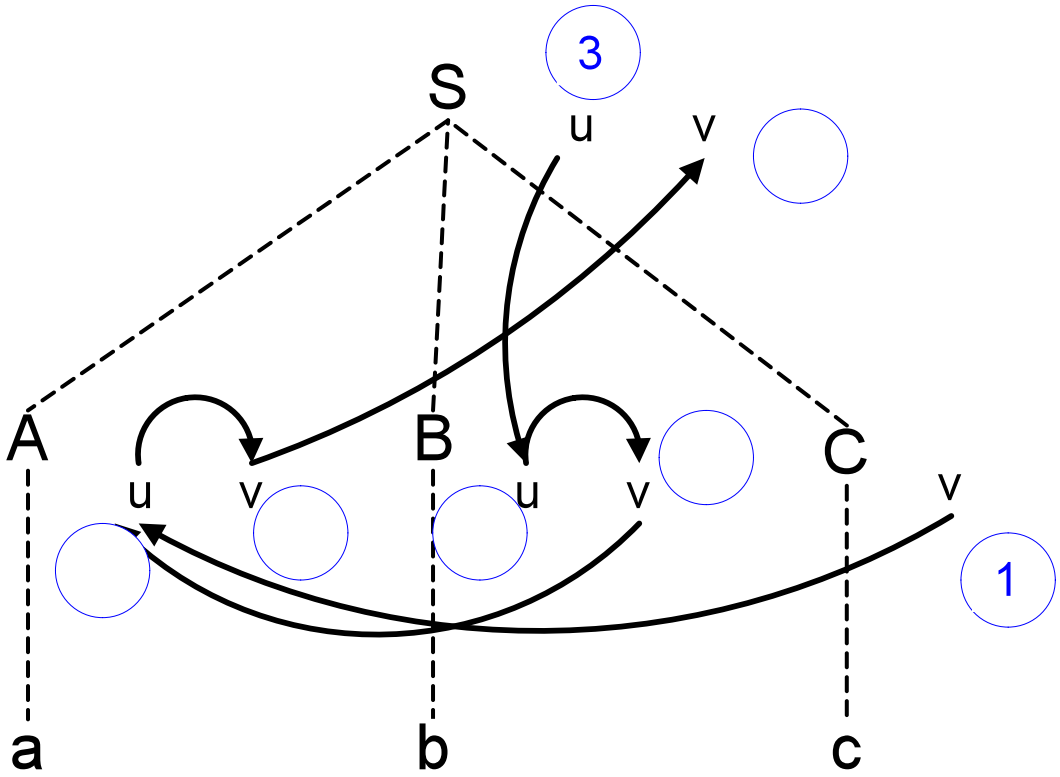

S
2
u
v
A
B
C
u
v
u
v
4
3
v
1
a
b
c

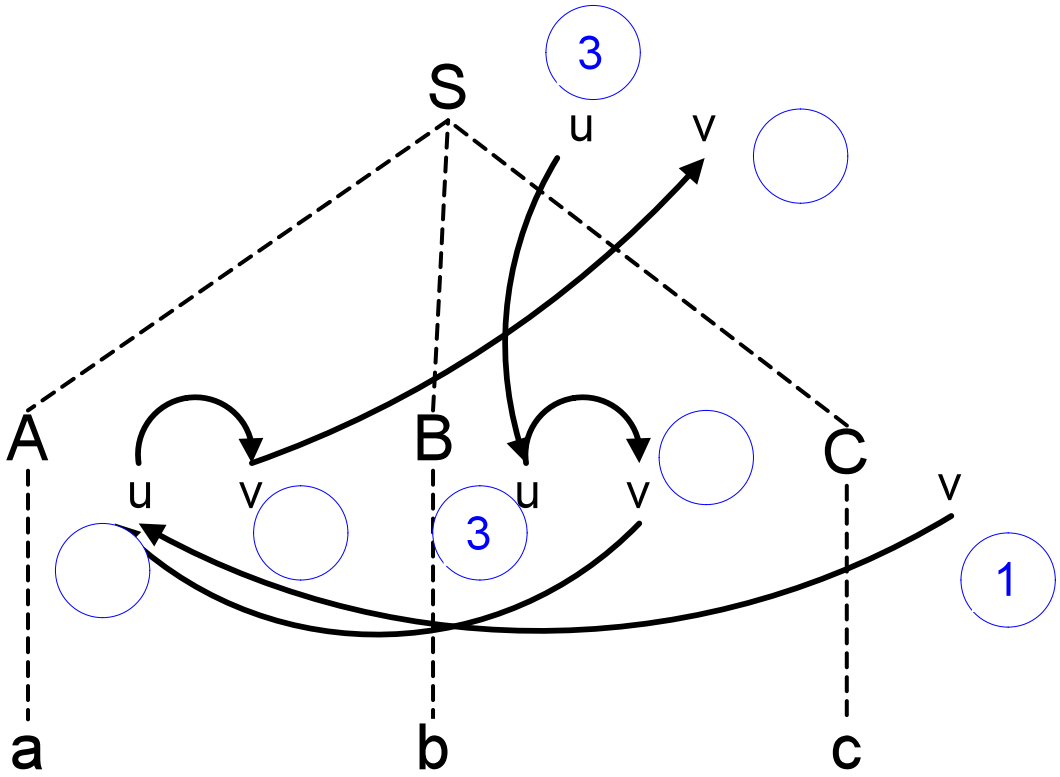

S
2
u
v
A
B
C
u
v
u
v
4
v
3
5
1
a
b
c

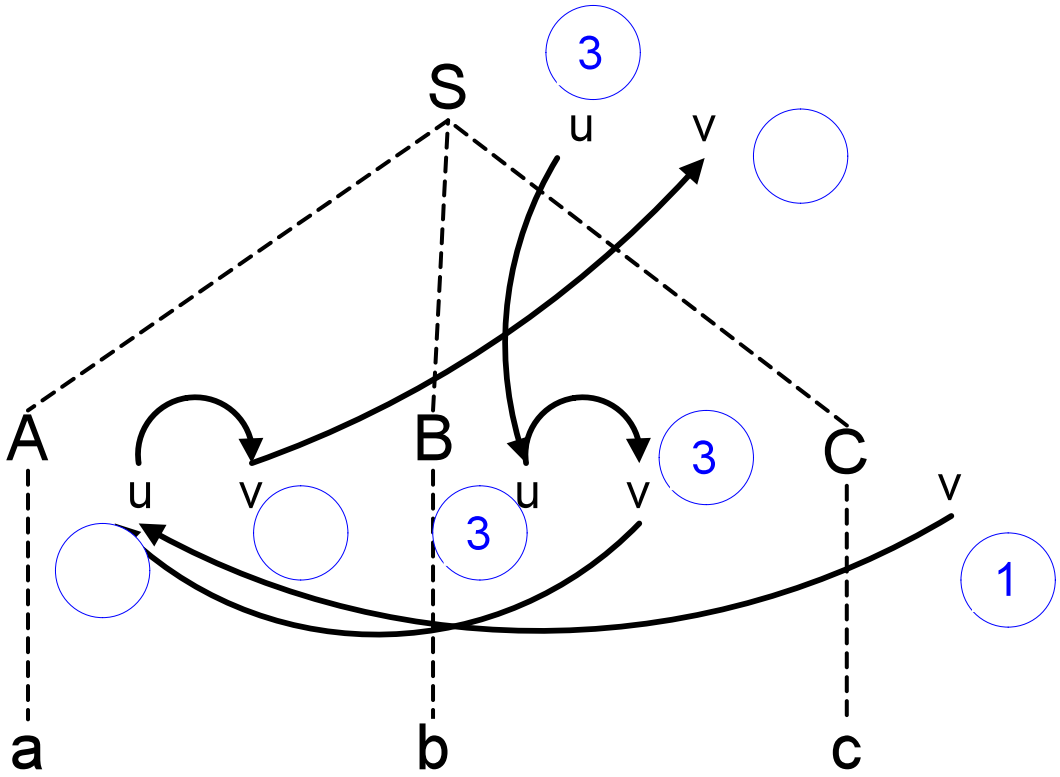

S
2
u
v
A
B
C
u
v
u
v
4
v
6
3
5
1
a
b
c

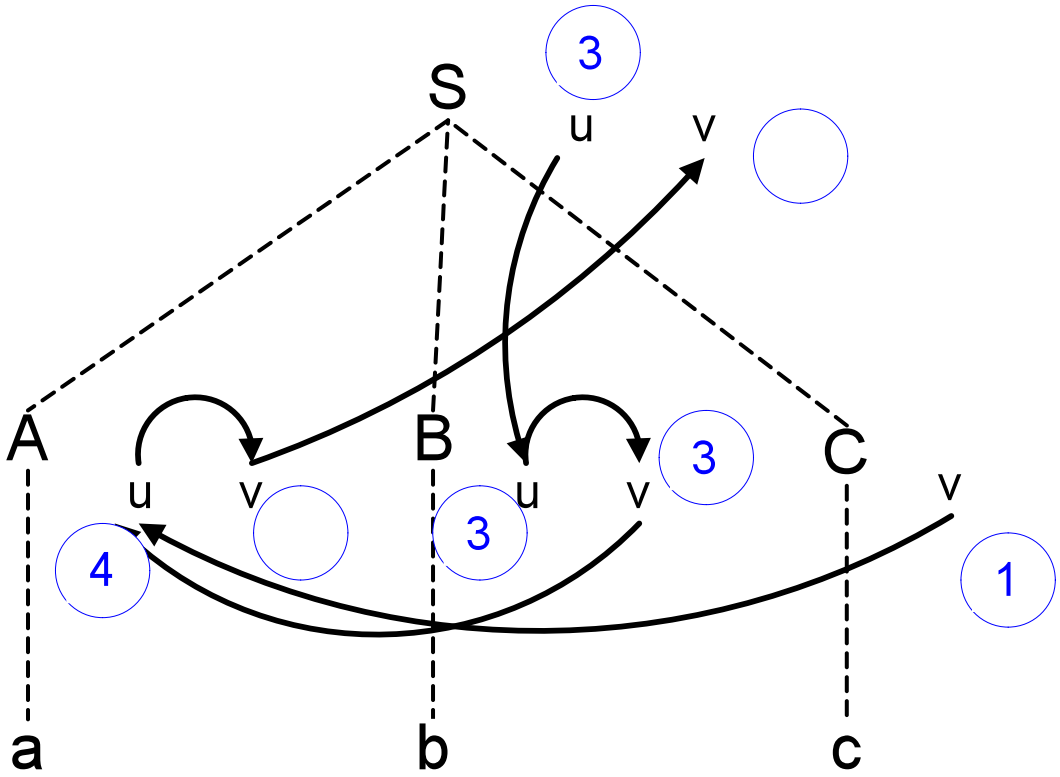

S
2
u
v
7
A
B
C
u
v
6
3
u
v
4
v
5
1
a
b
c

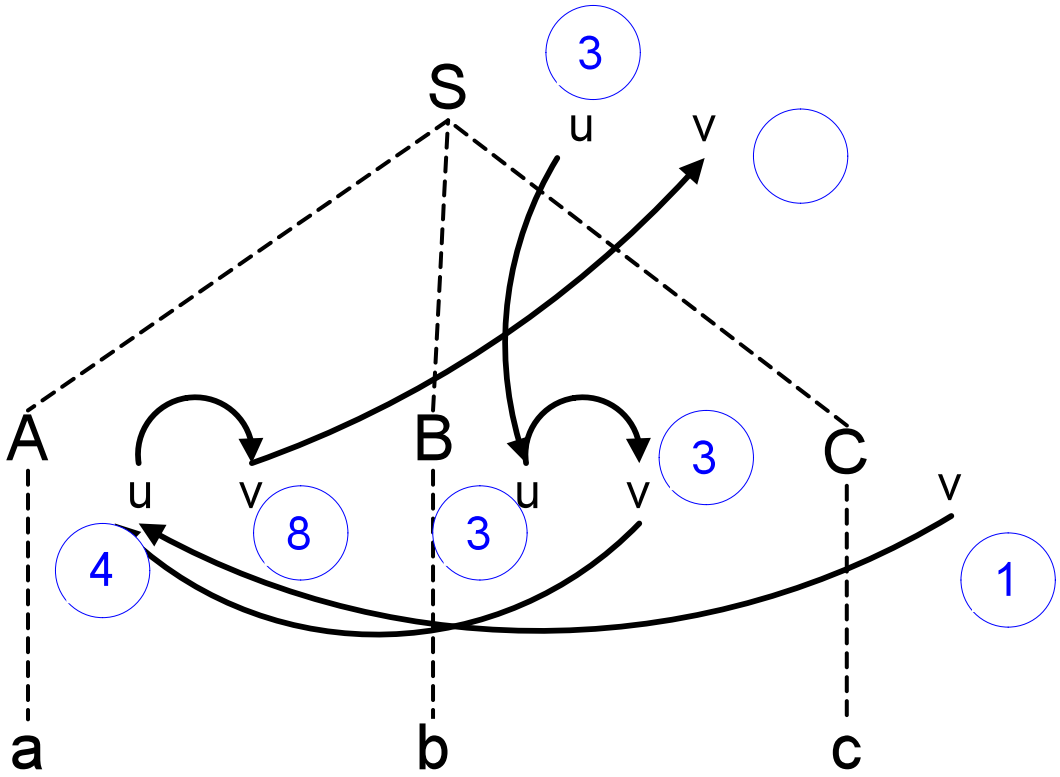
Oppgave 6.13 b)

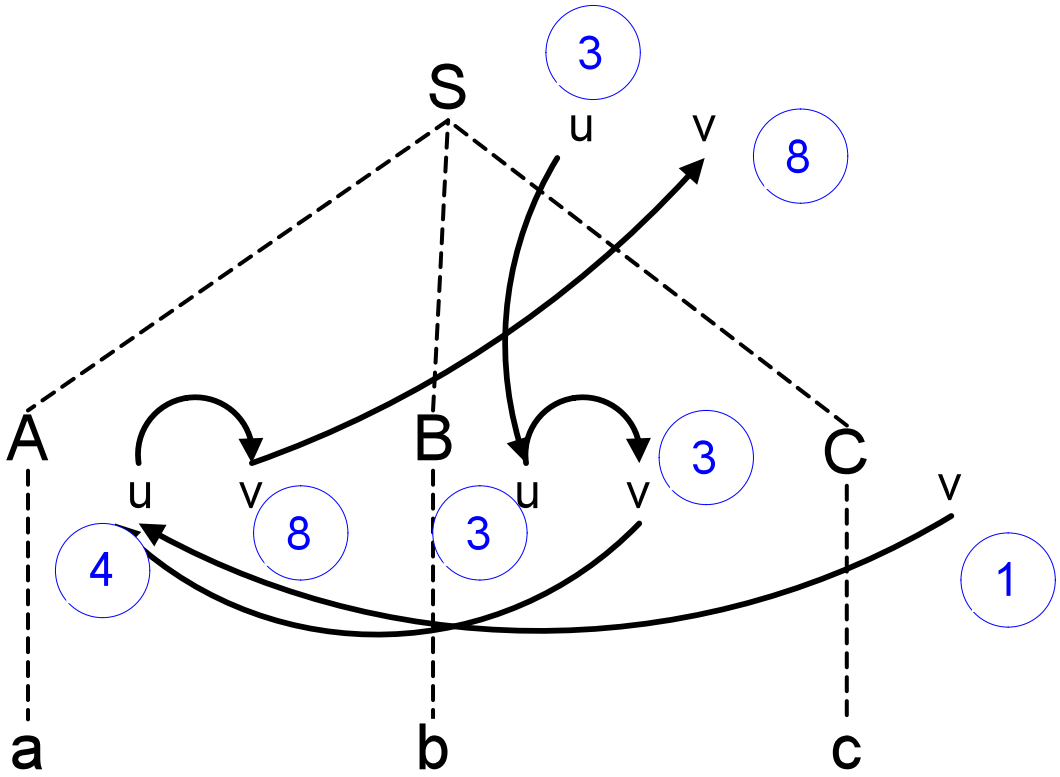



3
S
u
v
A
B
C
u
v
u
v
v
3
1
a
b
c


3
S
u
v
A
B
C
u
v
u
v
3
3
v
1
a
b
c


3
S
u
v
A
B
C
u
v
u
v
3
v
3
4
1
a
b
c


3
S
u
v
A
B
C
u
v
u
v
3
v
8
3
4
1
a
b
c

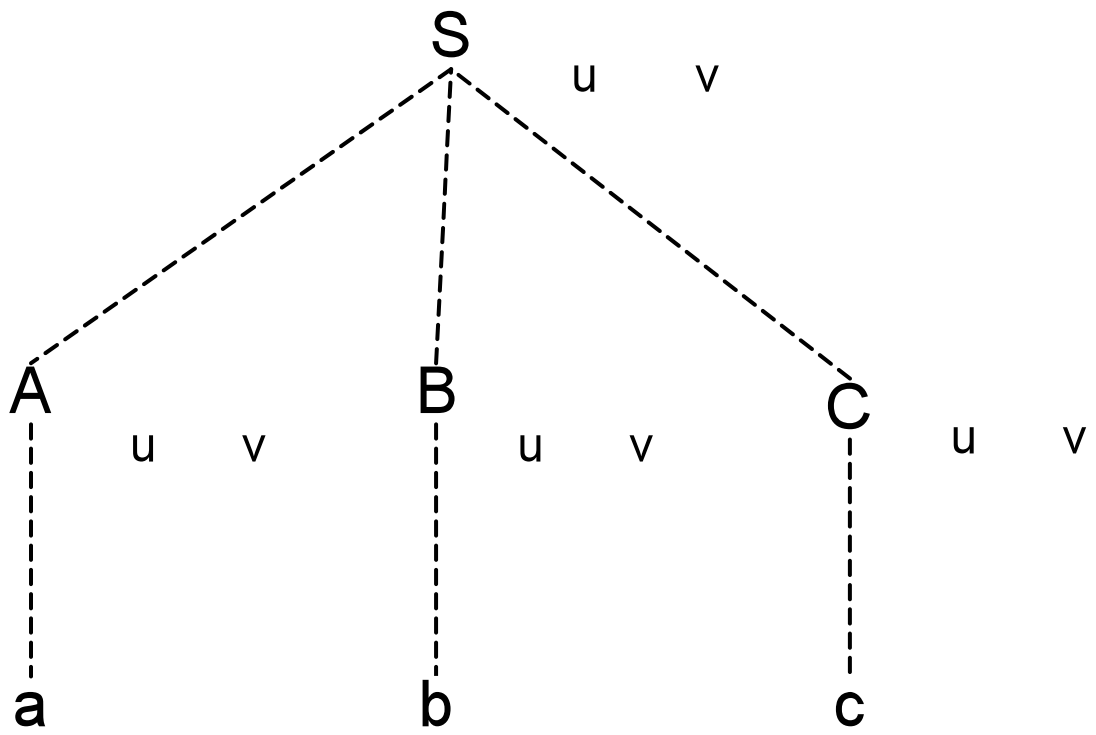

S
3
u
v
8
A
B
C
u
v
8
3
u
v
3
v
4
1
a
b
c

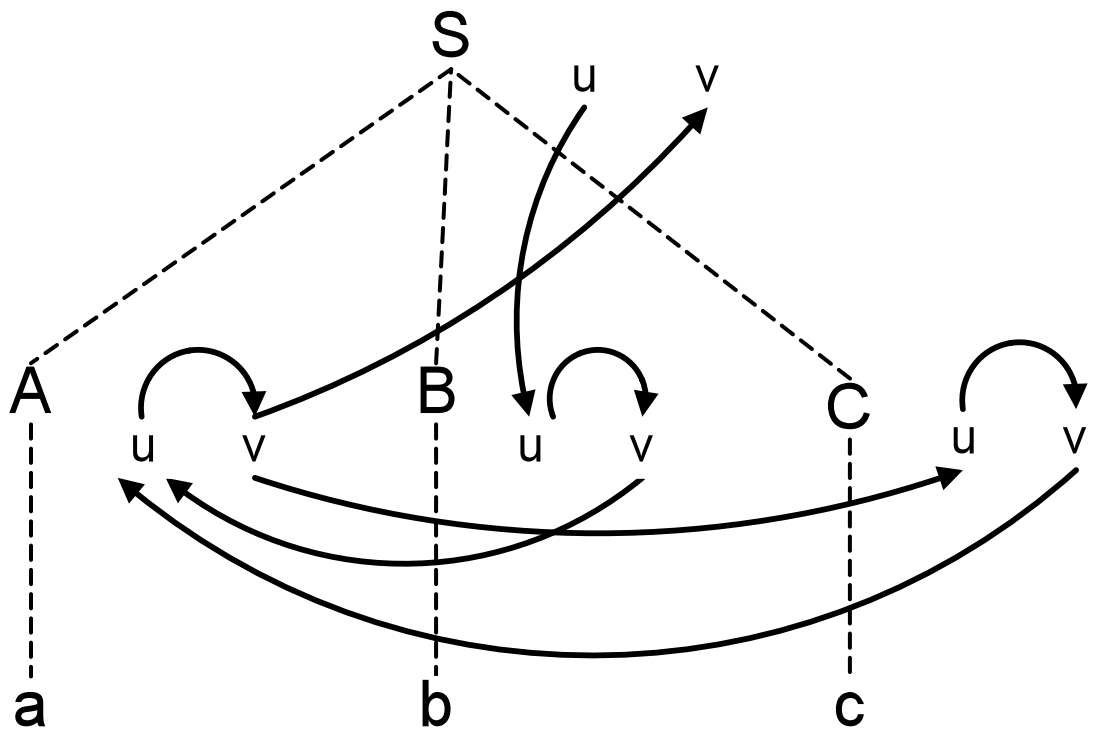
c. Suppose the attribute equations are modified as follows:

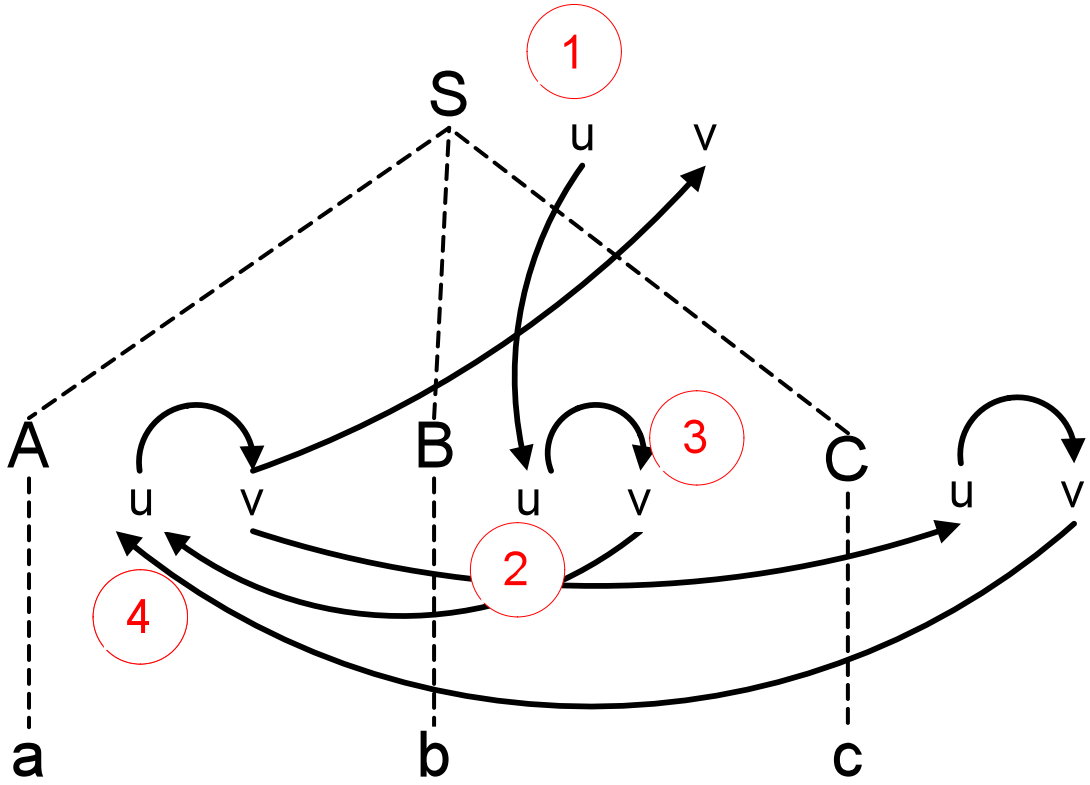
Grammar Rule	Semantic Rules
$S \rightarrow A\ B\ C$	$B.u = S.u$
	$C.u = A.v$
	$A.u = B.v + C.v$
	$S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = C.u - 2$

What value does $S.v$ have after attribute evaluation, if $S.u = 3$ before evaluation begins?

Oppgave 6.13 c)




S
u
v
A
B
C
u
v
u
v
u
v
a
b
c


1
S
u
v
A
B
C
3
u
v
u
v
u
v
2
4
a
b
c

```
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**	

Grammar Rule	Semantic Rule
class → **class** ***name*** **{** *decls* **}**	*decls*.*enclosingClassName* = ***name***.name
$decls_1$ → $decls_2$ **;** *decl*	$decls_2$.*enclosingClassName* = $decls_1$.*enclosingClassName* *decl*.*enclosingClassName* = $decls_1$.*enclosingClassName*
decls → *decl*	*decl*.*enclosingClassName* = *decls*.*enclosingClassName*
decl → *variable-decl*	
decl → *method-decl*	*method-decl*.*enclosingClassName* = *decl*.*enclosingClassName*
type → **int**	*type*.type = int
type → **bool**	*type*.type = bool
type → **void**	*type*.type = void

Grammar Rule	Semantic Rule
method-decl → *type* **name** **(** *params* **)** *body*	*method-decl*.OKconstructor = **if** (**name**.name = *method-decl.enclosingClassName*) **then if** (**not**(*type*.type = void)) **then** *error*("constructor not of type void") **else** *ok* Eller *method-decl*.OKconstructor = **if** (**name**.name = *method-decl.enclosingClassName*) **and** (**not**(*type*.type = void)) **then** *error*("constructor not of type void") **else** *ok*

10 - 2012

2007 – 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*. Besvar oppgaven ved å bruke vedlegg side 8.

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.kind = var* *parameter.type = type.type*
parameter → *type* **func id**	*parameter.kind = func* *parameter.type = type.type*
type → **int**	*type.type = integer*
type → **bool**	*type.type = boolean*
type → **void**	*type.type = void*

Grammar Rule	Semantic Rule
call → ***id*** *()*	*call.ok =* *(lookup(**id**.name).has_parameter=no)*
call → $\mathbf{id}_1$*(*$\mathbf{id}_2$*)*	*call.ok =* *(lookup(*$\mathbf{id}_1$*.name).has_parameter=yes)* *and* *(lookup(*$\mathbf{id}_2$*.name).kind=* *(lookup(*$\mathbf{id}_1$*.name).param-kind)* *and* *(lookup(*$\mathbf{id}_2$*.name).type=* *(lookup(*$\mathbf{id}_1$*.name).param-type)* *and* *(if lookup(*$\mathbf{id}_2$*.name).kind=func then* *(lookup(*$\mathbf{id}_2$*.name).has_parameter=no) else true)*

Eksamen 2010

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 side 7 slik at attributtet *ok* for *return-stmt* er `true` hvis typen til retur-uttryket *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*	*stmt-list*.type = *type*.type
type → **int**	*type*.type = Integer
type → **bool**	*type*.type = Boolean
$stmt\text{-}list_1 \rightarrow stmt\text{-}list_2\ stmt$	$stmt\text{-}list_2$.type = $stmt\text{-}list_1$.type *stmt*.type = $stmt\text{-}list_1$.type
stmt-list → *stmt*	*stmt*.type = *stmt-list*.type
stmt → *return-stmt*	*return-stmt*.type = *stmt*.type
return-stmt → **return** *exp*	*return-stmt*.ok = (*return-stmt*.type = *exp*.type)

Grammar Rule	Semantic Rule
exp → *id*	*exp*.type = lookup(*id.name*)
exp → id_1 **+** id_2	*exp*.type = if lookup(id_1.*name*)= Integer and lookup(id_2.*name*)= Integer then Integer else ErrorType
exp → **true**	*exp*.type = Boolean
exp → **false**	*exp*.type = Boolean