

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
$exp_1 \rightarrow exp_2 + term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2 - term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2 * factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow (exp)$	$factor.val = exp.val$
$factor \rightarrow \mathbf{number}$	$factor.val = \mathbf{number.val}$

Oppgave 6.5

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

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

$$decl \rightarrow var\text{-list} : type$$
$$var\text{-list} \rightarrow var\text{-list} , id \mid id$$
$$type \rightarrow \mathbf{integer} \mid \mathbf{real}$$

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
```

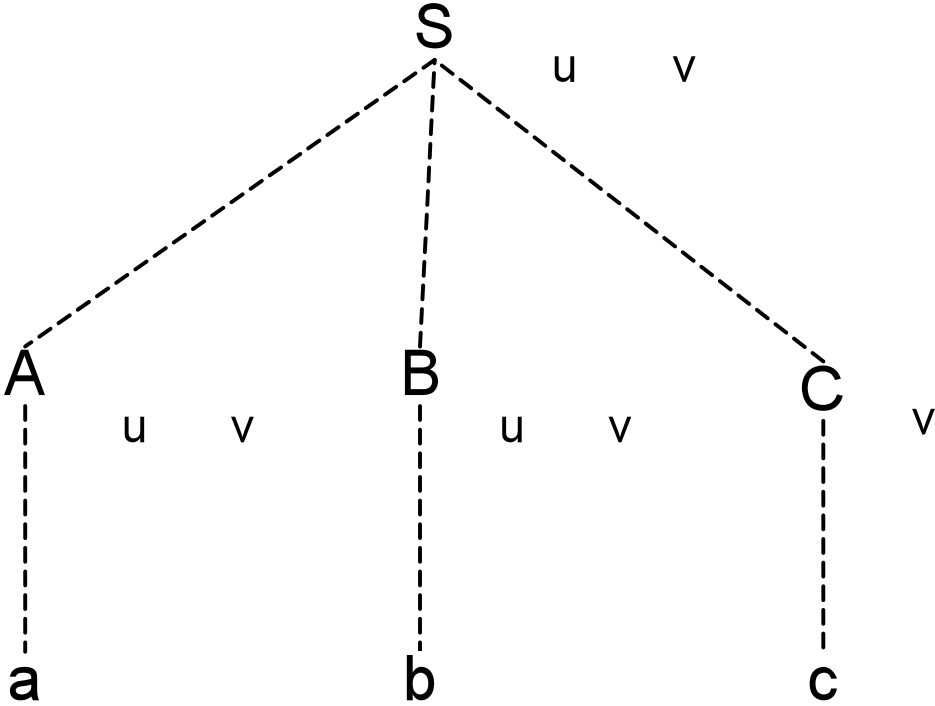
Grammar Rule	Semantic Rule
<i>decl</i> → <i>var-list</i> : <i>type</i>	<i>var-list.dtype</i> = <i>type.dtype</i>
<i>var-list</i> ₁ → <i>var-list</i> ₂ , <i>id</i>	<i>var-list</i> _{2.dtype} = <i>var-list</i> _{1.dtype} <i>id.dtype</i> = <i>var-list</i> _{1.dtype}
<i>var-list</i> → <i>id</i>	<i>id.dtype</i> = <i>var-list.dtype</i>
<i>type</i> → int	<i>type.dtype</i> = <i>integer</i>
<i>type</i> → real	<i>type.dtype</i> = <i>real</i>

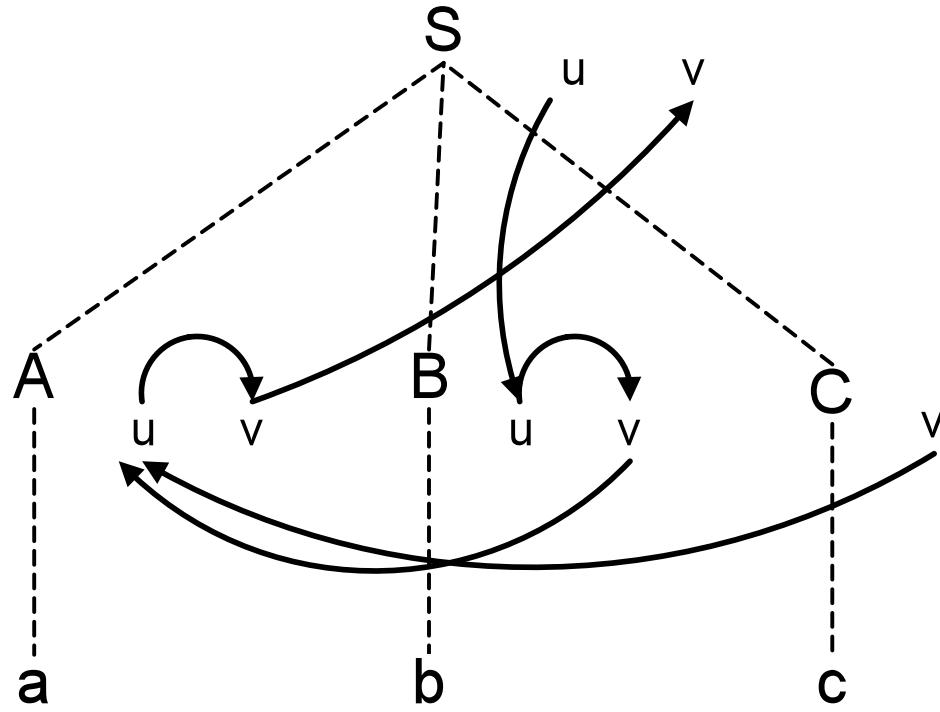
6.13 Consider the following attribute grammar:

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

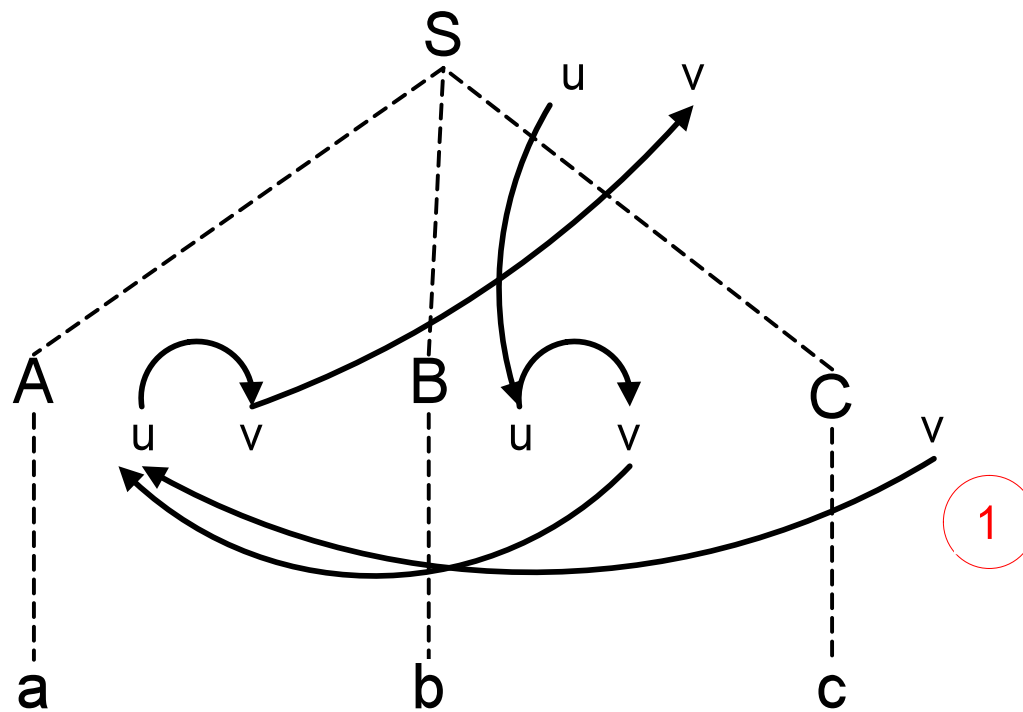
- Draw the parse tree for the string abc (the only string in the language), and draw the dependency graph for the associated attributes. Describe a correct order for the evaluation of the attributes.
- Suppose that $S.u$ is assigned the value 3 before attribute evaluation begins. What is the value of $S.v$ when evaluation has finished?

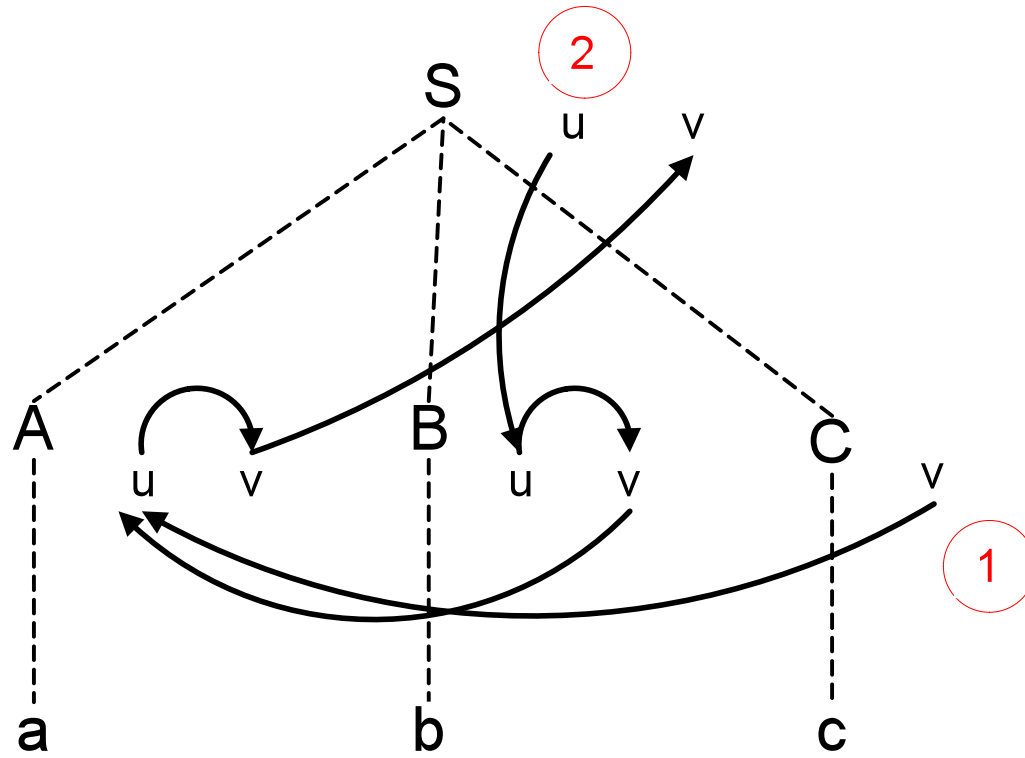
Oppgave 6.13 a 1)

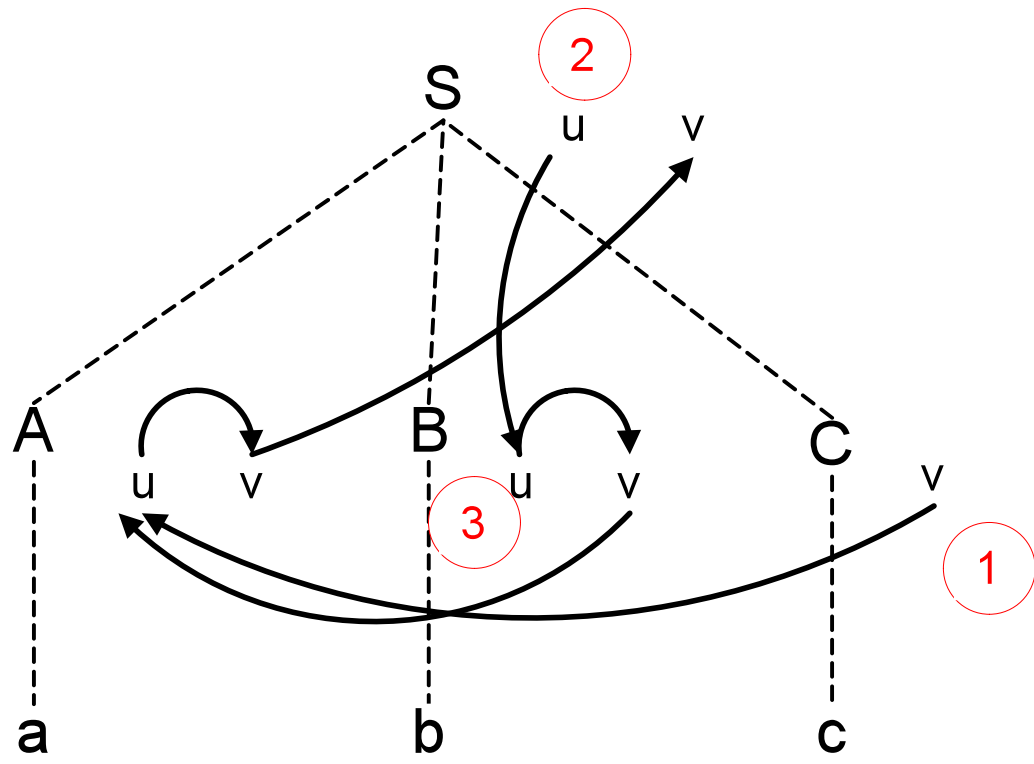


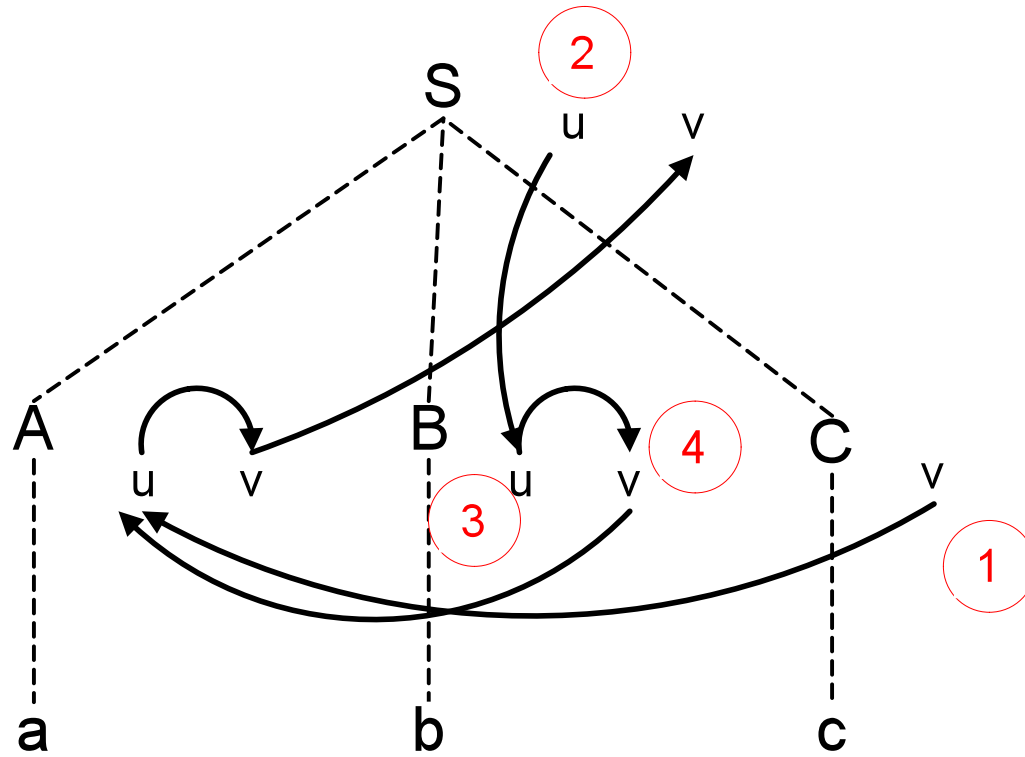


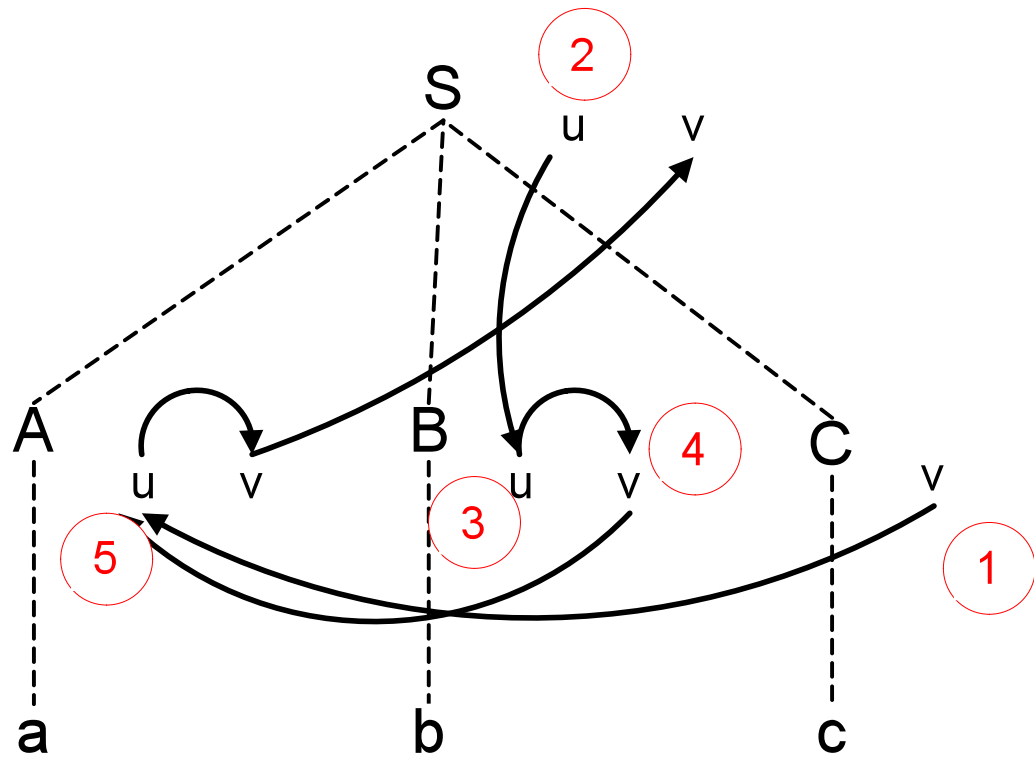
Oppgave 6.13 a 2)

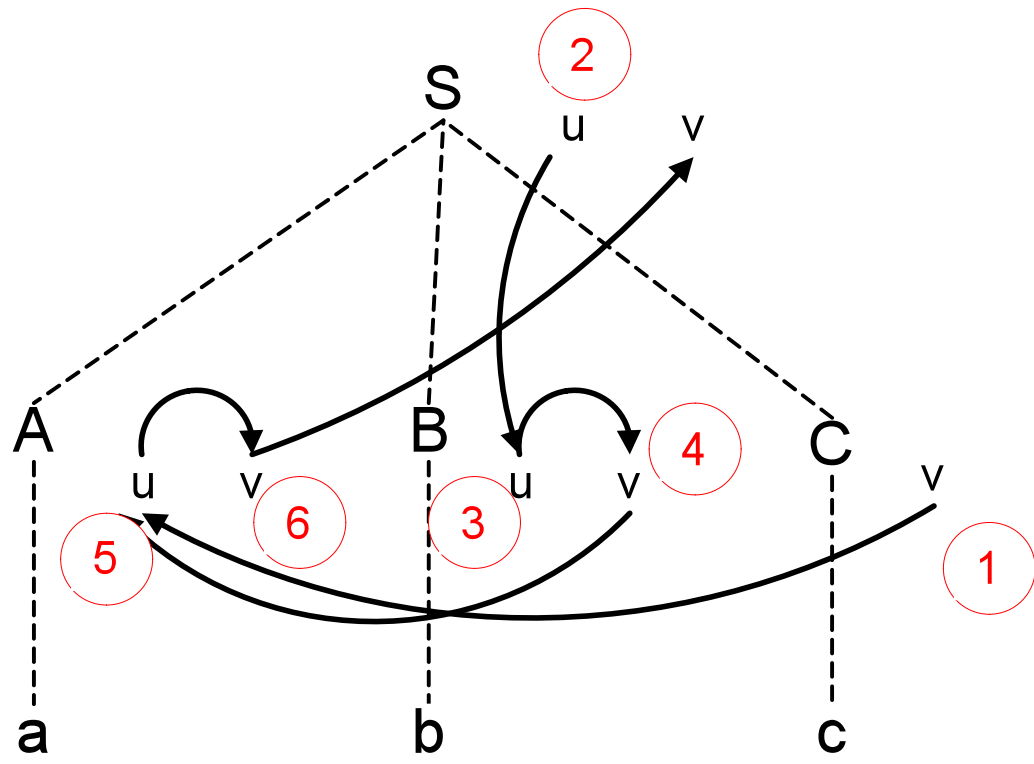


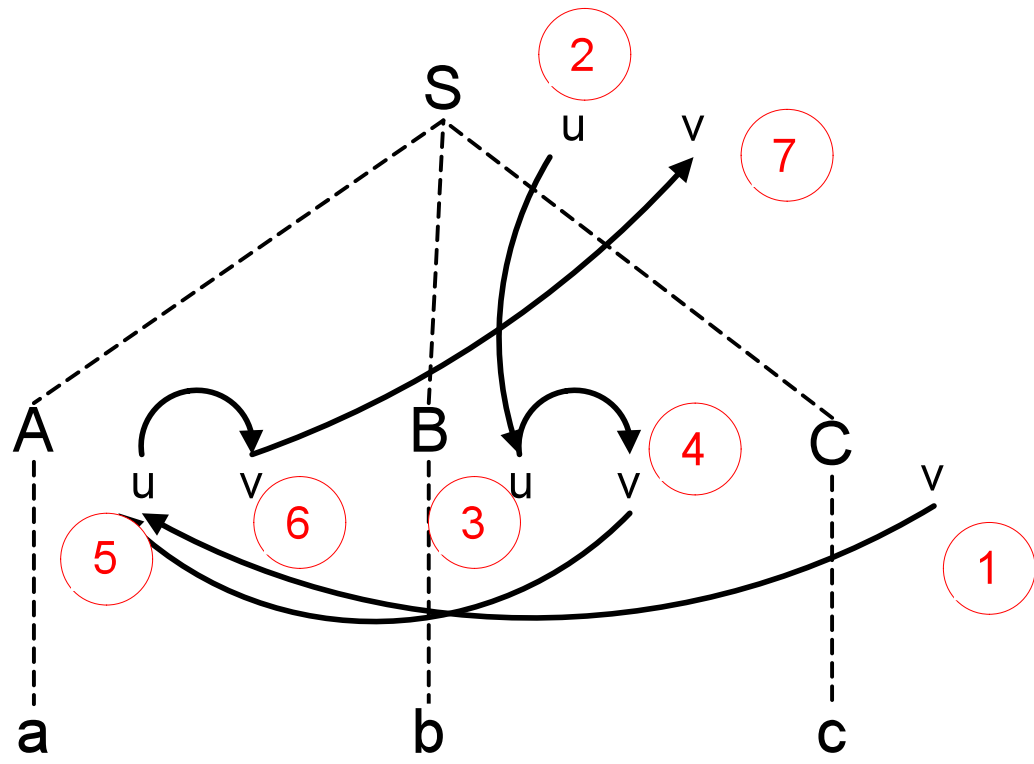




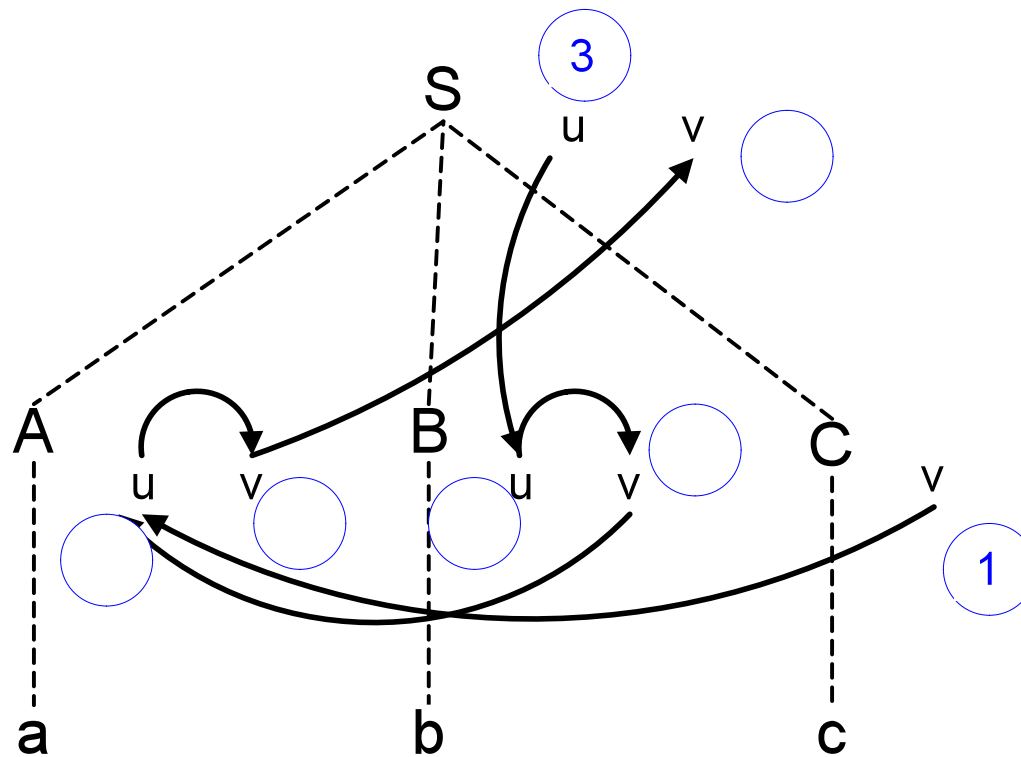


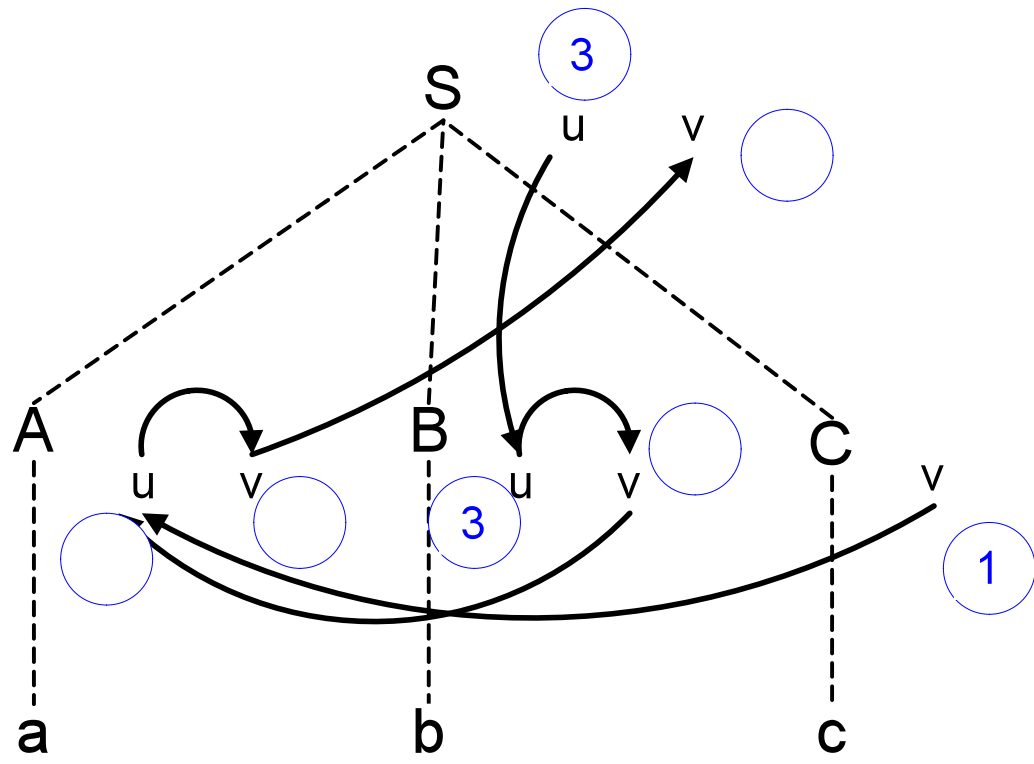


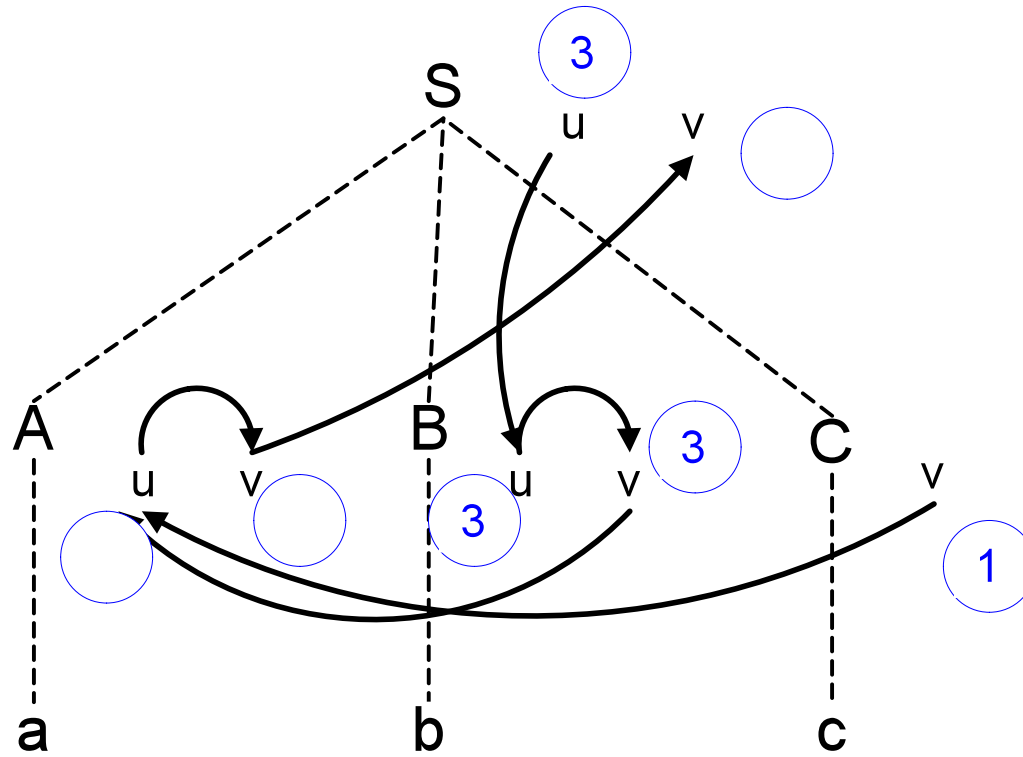


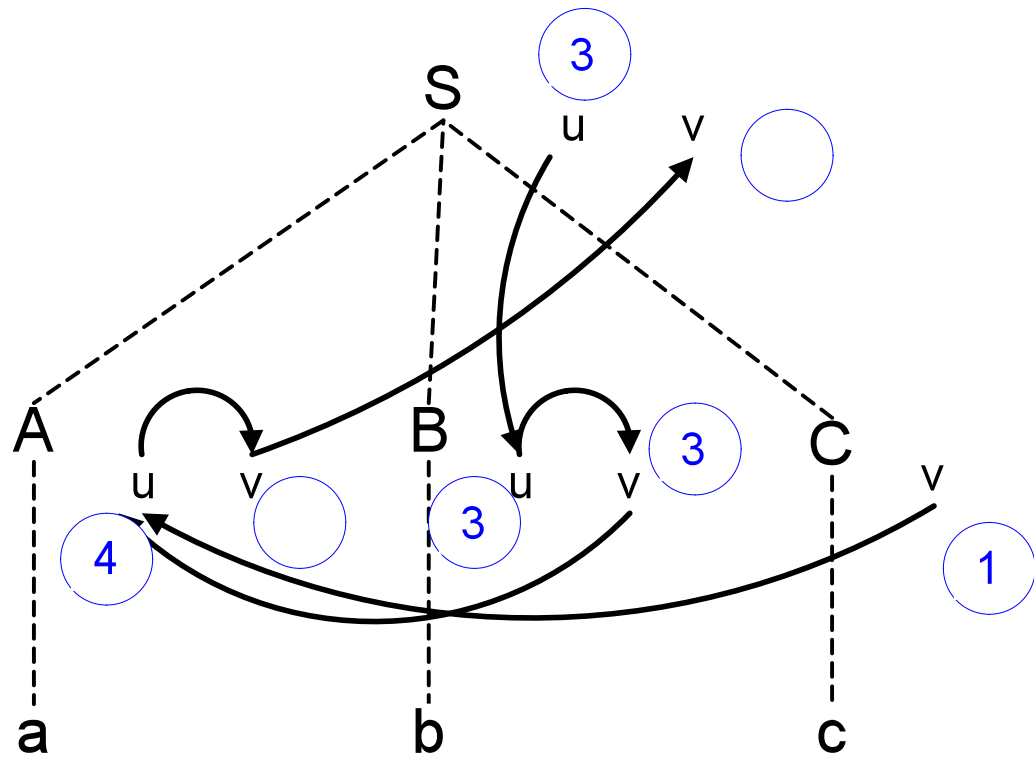


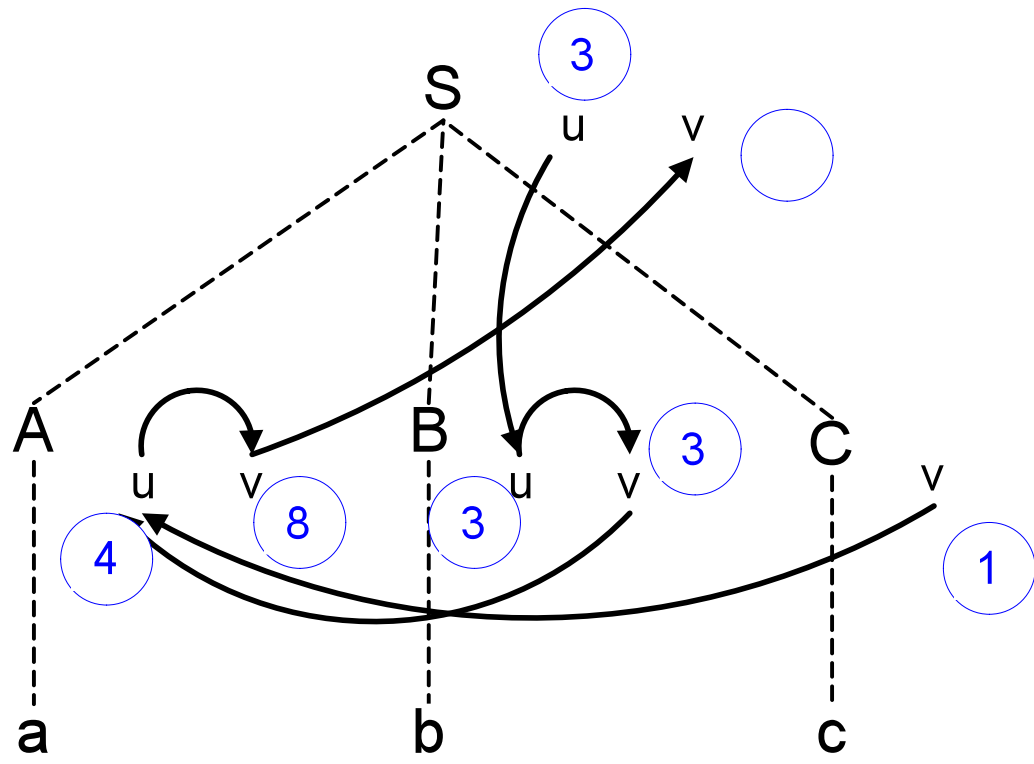
Oppgave 6.13 b)

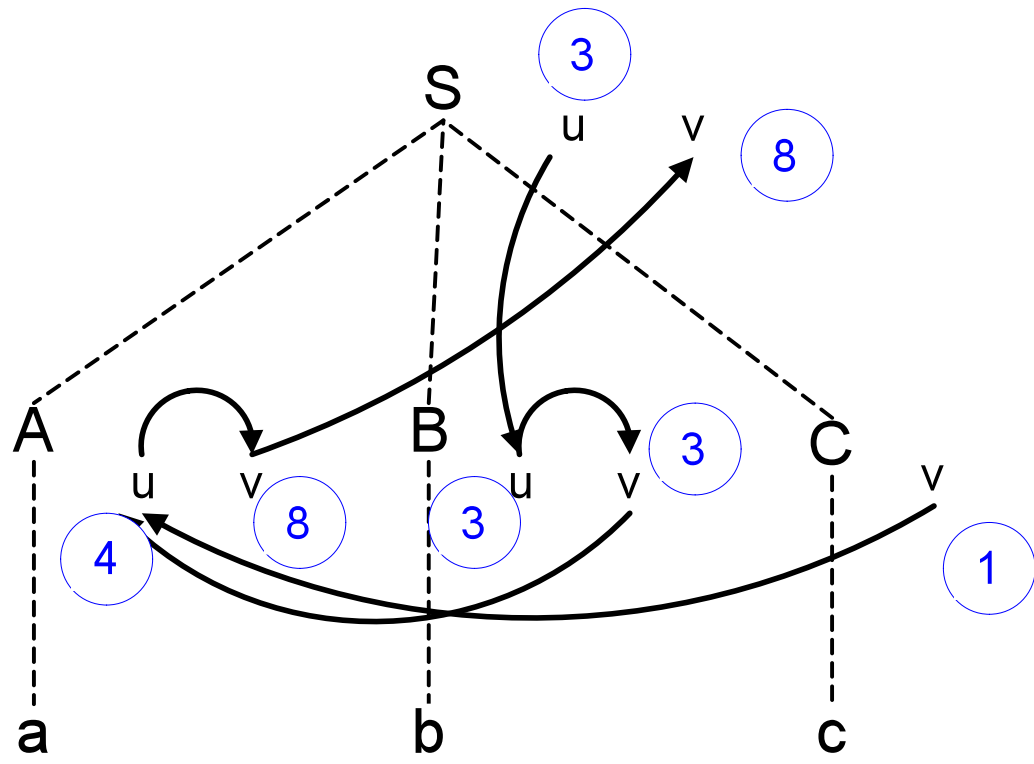










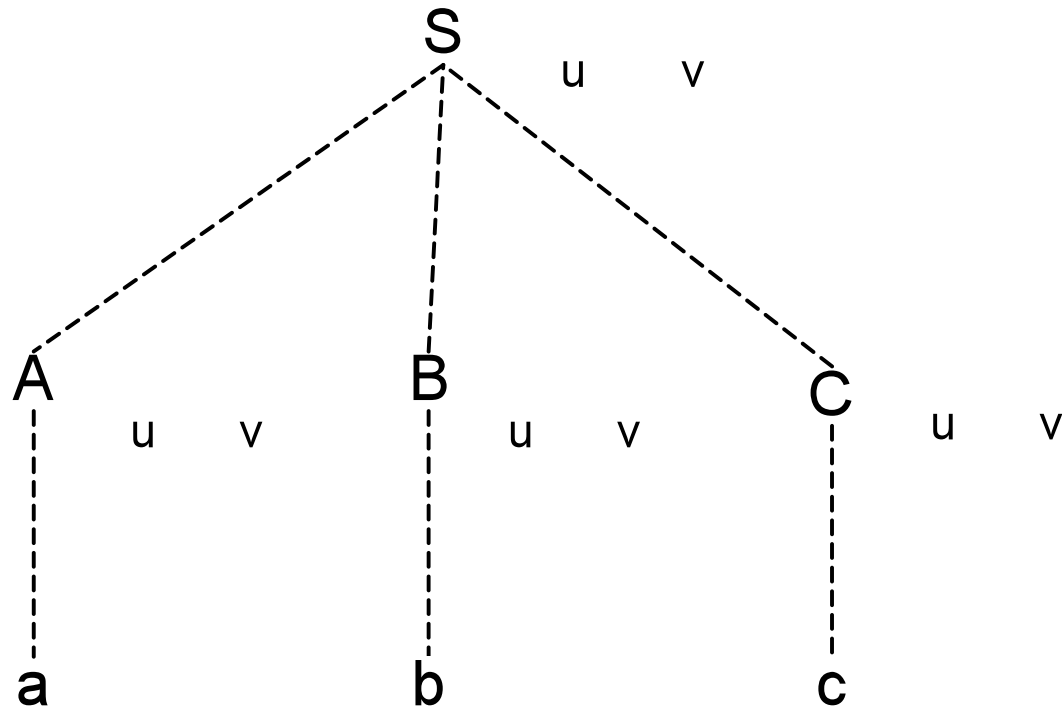


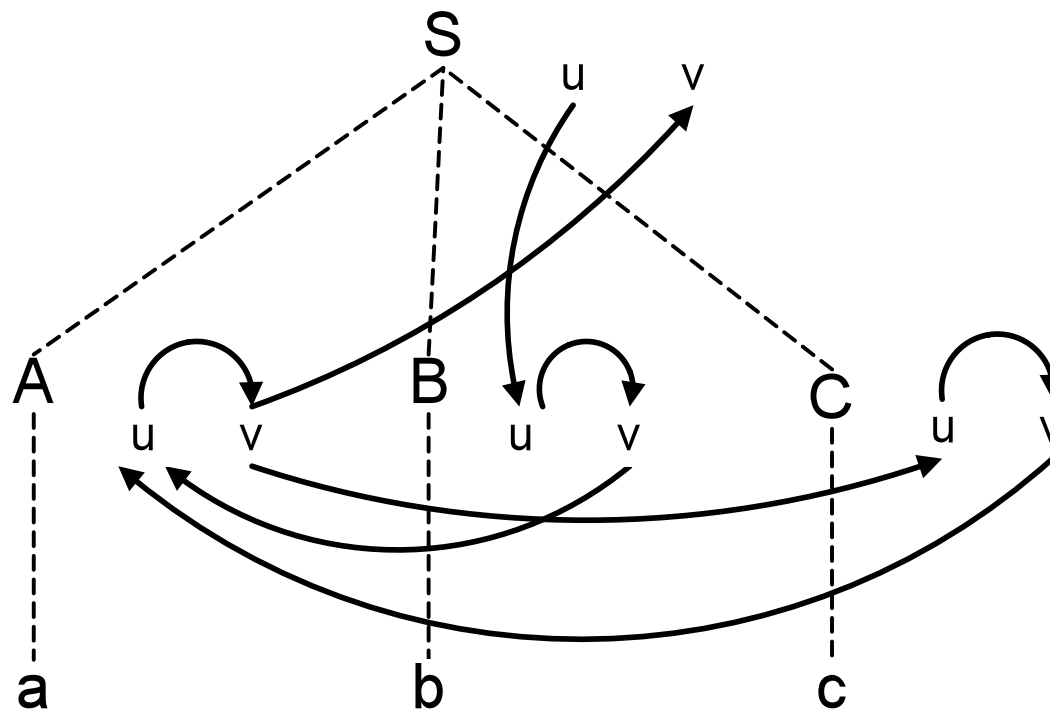
c. Suppose the attribute equations are modified as follows:

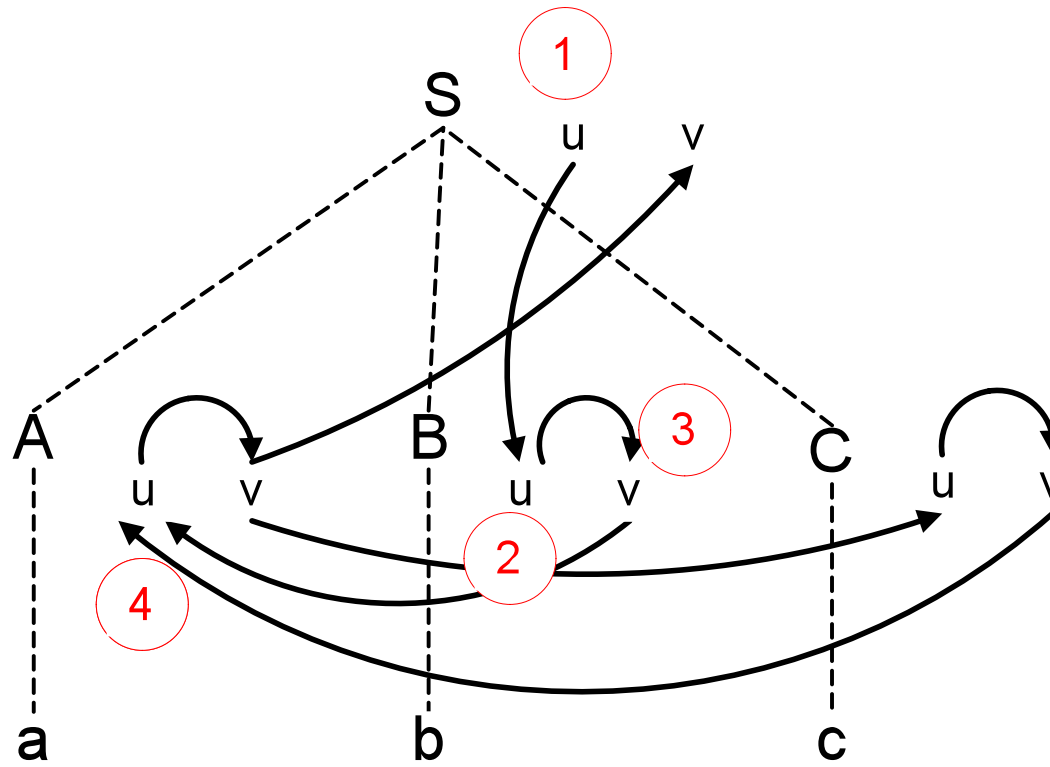
Grammar Rule	Semantic Rules
$S \rightarrow A B C$	$B.u = S.u$ $C.u = A.v$ $A.u = B.v + C.v$ $S.v = A.v$
$A \rightarrow a$	$A.v = 2 * A.u$
$B \rightarrow b$	$B.v = B.u$
$C \rightarrow c$	$C.v = C.u - 2$

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

Oppgave 6.13 c)








```
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
<code>class → class name { decls }</code>	
<code>decls → decls ; decl</code>	
<code>decls → decl</code>	
<code>decl → variable-decl</code>	<i>Skal ikke fylles ut</i>
<code>decl → method-decl</code>	
<code>method-decl → type name (params) body</code>	
<code>type → int</code>	
<code>type → bool</code>	
<code>type → void</code>	

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

Grammar Rule	Semantic Rule
<pre>method-decl → type name (params) body</pre>	<pre>if (name.name = method-decl.enclosingClassName) then if (not(type.type = void)) then error("constructor not of type void") Eller if (name.name = method- decl.enclosingClassName) and (not(type.type = void)) then error("constructor not of type void")</pre>

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 deklarerert (du kan altså anta at det allerede er gjort ved andre mekanismer).

Grammar Rule	Semantic Rule
<i>function-decl</i> → type id () <i>body</i>	<i>function-decl.has_parameter</i> = no
<i>function-decl</i> → type id (<i>parameter</i>) <i>body</i>	<i>function-decl.has_parameter</i> = yes <i>function-decl.param-kind</i> = <i>parameter.kind</i> <i>function-decl.param-type</i> = <i>parameter.type</i>
<i>parameter</i> → type id	<i>parameter.kind</i> = var <i>parameter.type</i> = <i>type.type</i>
<i>parameter</i> → type func id	<i>parameter.kind</i> = func <i>parameter.type</i> = <i>type.type</i>
<i>type</i> → int	<i>type.type</i> = integer
<i>type</i> → bool	<i>type.type</i> = boolean
<i>type</i> → void	<i>type.type</i> = void

Grammar Rule	Semantic Rule
<code>call → id ()</code>	<code>call.ok = (lookup(id.name).has_parameter=no)</code>
<code>call → id₁(id₂)</code>	<code>call.ok = (lookup(id₁.name).has_parameter=yes) and (lookup(id₂.name).kind= (lookup(id₁.name).param-kind) and (lookup(id₂.name).type= (lookup(id₁.name).param-type) and (if lookup(id₂.name).kind=func then (lookup(id₂.name).has_parameter=no) else true)</code>

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 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 variabelen med navnet *id.name*.

Grammar Rule	Semantic Rule
$func \rightarrow type \mathbf{func} id signature$ $stmt-list$	$stmt-list.type = type.type$
$type \rightarrow \mathbf{int}$	$type.type = Integer$
$type \rightarrow \mathbf{bool}$	$type.type = Boolean$
$stmt-list_1 \rightarrow stmt-list_2 stmt$	$stmt-list_2.type = stmt-list_1.type$ $stmt.type = stmt-list_1.type$
$stmt-list \rightarrow stmt$	$stmt.type = stmt-list.type$
$stmt \rightarrow return-stmt$	$return-stmt.type = stmt.type$
$return-stmt \rightarrow \mathbf{return} exp$	$return-stmt.ok =$ $(return-stmt.type = exp.type)$

Grammar Rule	Semantic Rule
$exp \rightarrow id$	$exp.type = lookup(id.name)$
$exp \rightarrow id_1 + id_2$	$exp.type =$ if $lookup(id_1.name) = Integer$ and $lookup(id_2.name) = Integer$ then $Integer$ else $ErrorType$
$exp \rightarrow \mathbf{true}$	$exp.type = Boolean$
$exp \rightarrow \mathbf{false}$	$exp.type = Boolean$