Semantic analysis – syntax-directed definitions

Semantic Analysis

This is the 3rd stage in the compilation process

A program that is lexically and syntactically correct, may still fail to compile.

Some familiar examples:

```
void main()
   {int a; a = a + b}
```

Use of undeclared *b*

```
void foo()
   {int a, b;  double a;  a = a + b}
```

Duplicated declaration of *a*

Semantic analysis - goals

But why objects (variables, constants, functions, classes, etc.) declarations are needed at all?

Help to:

- Decide on the size of memory needed to keep values of the object
- **Check that the way the object is used (operations on the object, access to its elements, …) fits its declaration**

  This is the **main goal** of semantic analysis

Examples of inconsistency between object declaration and use

- assignment to a constant `const int length = 10; … ; length = 15;`
- non-integer value of index in array element `arr[2.45]`
- attempt to access a field of object that was not declared as structure

  `int x; … ; x.field = 5;`
- function call with a wrong amount of parameters

  `void do_smthng(int a, real b); … ; do_smthng(5, 3.14, 2020)`
- inconsistency of types between the left and right sides of assignment

  `int i_x; double d_y; … ; i_x = d_y + 2.785;`

<u>And much more</u> – depends on semantic rules of specific language

Do we need a new mechanism to address the above problems?

Idea:

- define the language grammar that allows to derive only programs that are semantically correct

- if parser accepts an input, then this input is correct both syntactically and semantically

- we already know how to construct a parser !!!

Nice idea, but…
This is impossible !

Reason

- grammars used to define syntax rules are context-free
- BUT: programming languages are <u>not</u> context free
  semantic rules <u>can't be expressed</u> by a context-free grammar

<u>Conclusion</u>

- **need to add to the language grammar some actions**
  in order to check that semantic rules are respected

How this can be done?

Example

$L = \{a^n b^n c^n \mid n \geq 0\}$ L is **not** context-free

Consider grammar G:

$S \rightarrow A\ B\ C$

$A \rightarrow aA$

All words from L can be derived in G! Example:

$A \rightarrow \varepsilon$

$S \rightarrow ABC \rightarrow aABC \rightarrow aBC \rightarrow abBC \rightarrow abC \rightarrow abcC \rightarrow abc$

$B \rightarrow bB$

$B \rightarrow \varepsilon$

$C \rightarrow cC$

$C \rightarrow \varepsilon$

Example

$L = \{a^n b^n c^n \mid n \geq 0\}$ L is **not** context-free

Consider grammar G:

$S \rightarrow A\ B\ C$

$A \rightarrow aA$

$A \rightarrow \varepsilon$

$B \rightarrow bB$

$B \rightarrow \varepsilon$

$C \rightarrow cC$

$C \rightarrow \varepsilon$

All words from L can be derived in G! Example:

$S \rightarrow ABC \rightarrow aABC \rightarrow aBC \rightarrow abBC \rightarrow abC \rightarrow abcC \rightarrow abc$

But: L(G) != L ; L(G) = {a*b*c*}

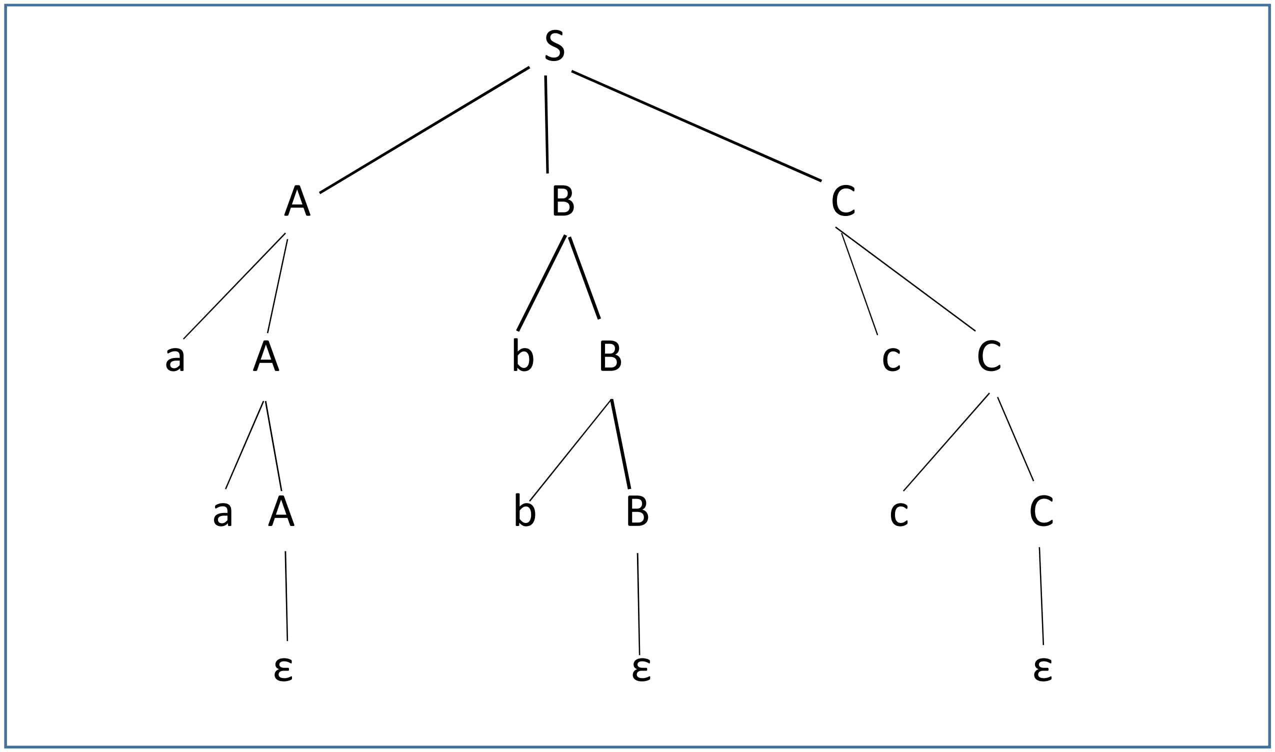
How to guarantee the needed balance of tokens a, b, c ?

Idea: count occurrences of a, b, c in the derived input; check that values of all 3 counters are equal

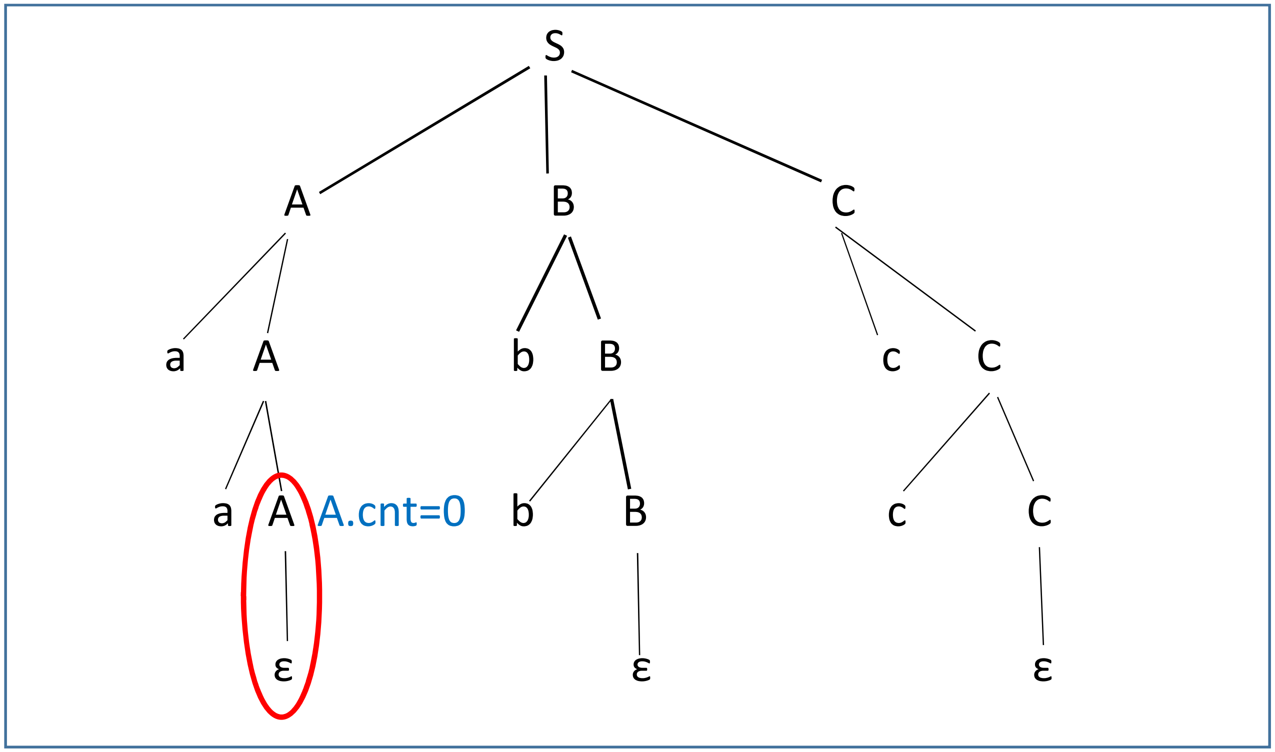
Implementation

- use counters A.cnt, B.cnt, C.cnt (# of tokens derived from the relevant variable)
- add counting actions to the grammar rules

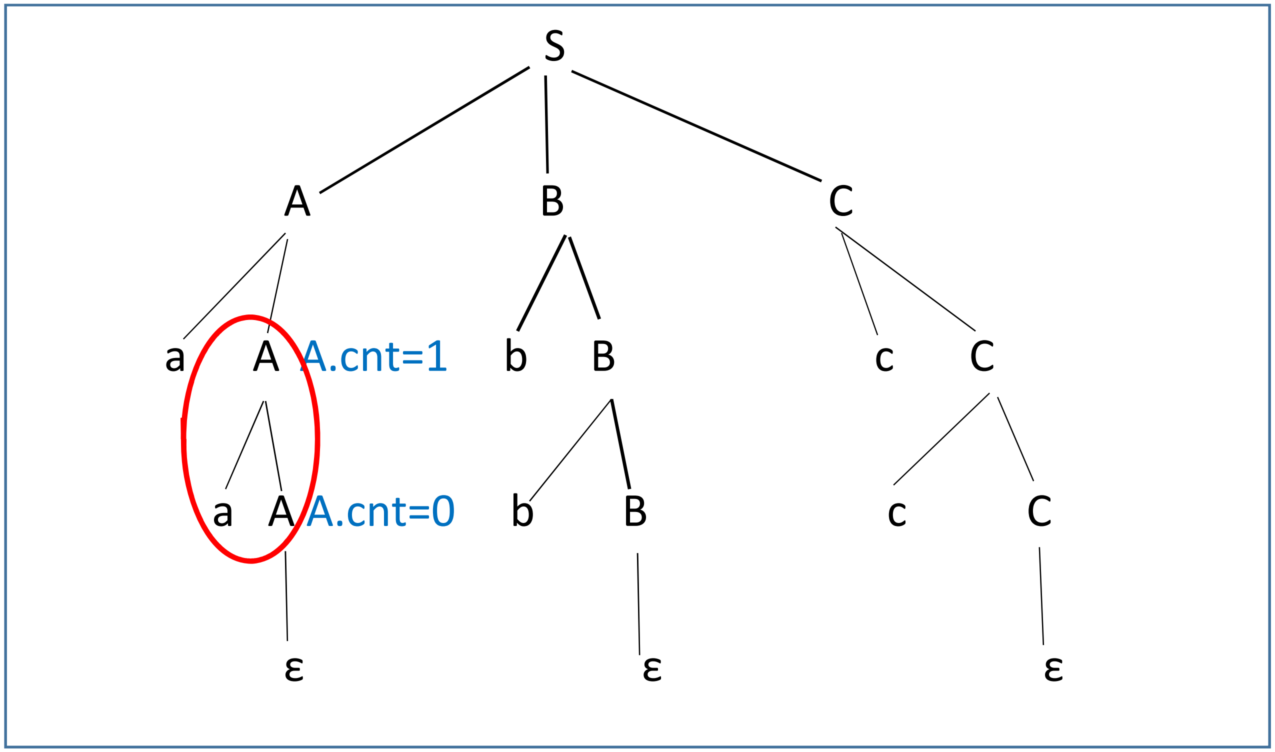
- $S \rightarrow A\ B\ C$ { **if** (A.cnt == B.cnt == C.cnt) **then** print(GOOD) **else** print(BAD) }
- $A \rightarrow aA_1$ { $A.cnt = A_1.cnt + 1$ } **distinguish between**
- $A \rightarrow \varepsilon$ { $A.cnt = 0$ }
- $B \rightarrow bB_1$ { $B.cnt = B_1.cnt + 1$ } **different occurrences**
- $B \rightarrow \varepsilon$ { $B.cnt = 0$ }
- $C \rightarrow cC_1$ { $C.cnt = C_1.cnt + 1$ } **of a variable in same rule**
- $C \rightarrow \varepsilon$ { $C.cnt = 0$ }



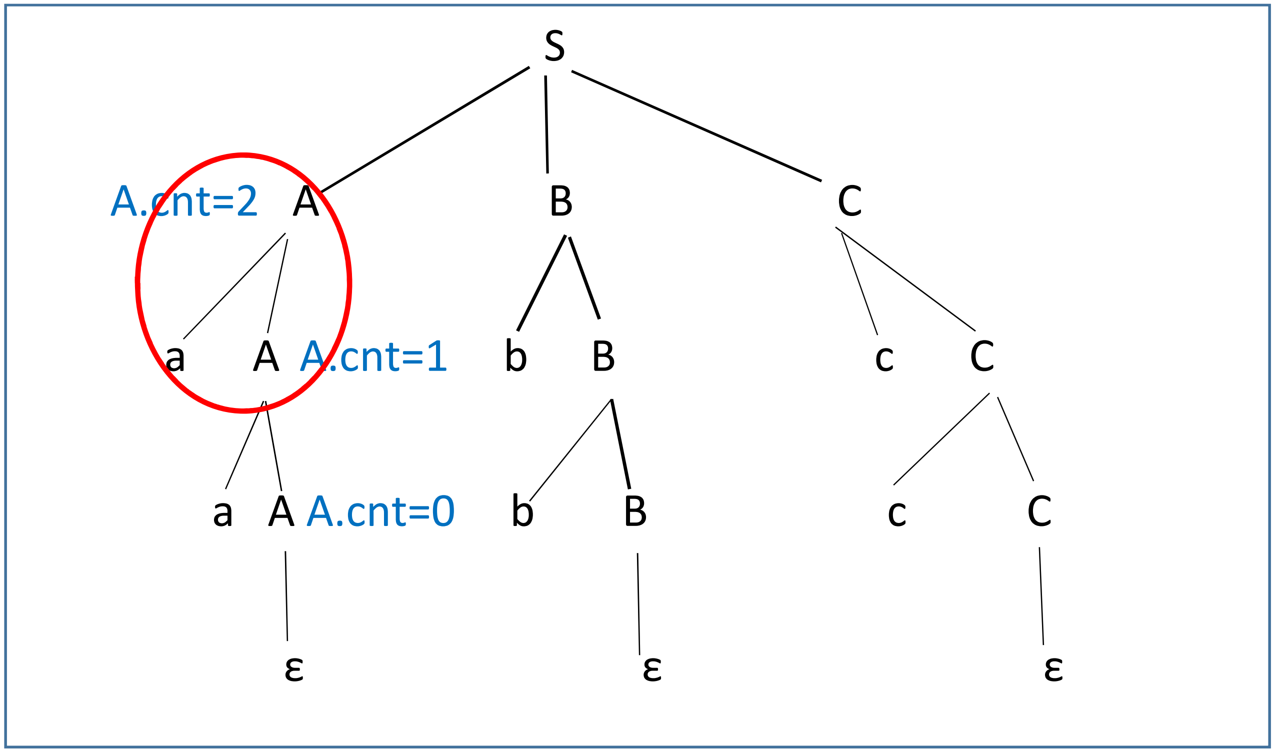
Derivation tree for input aabbcc



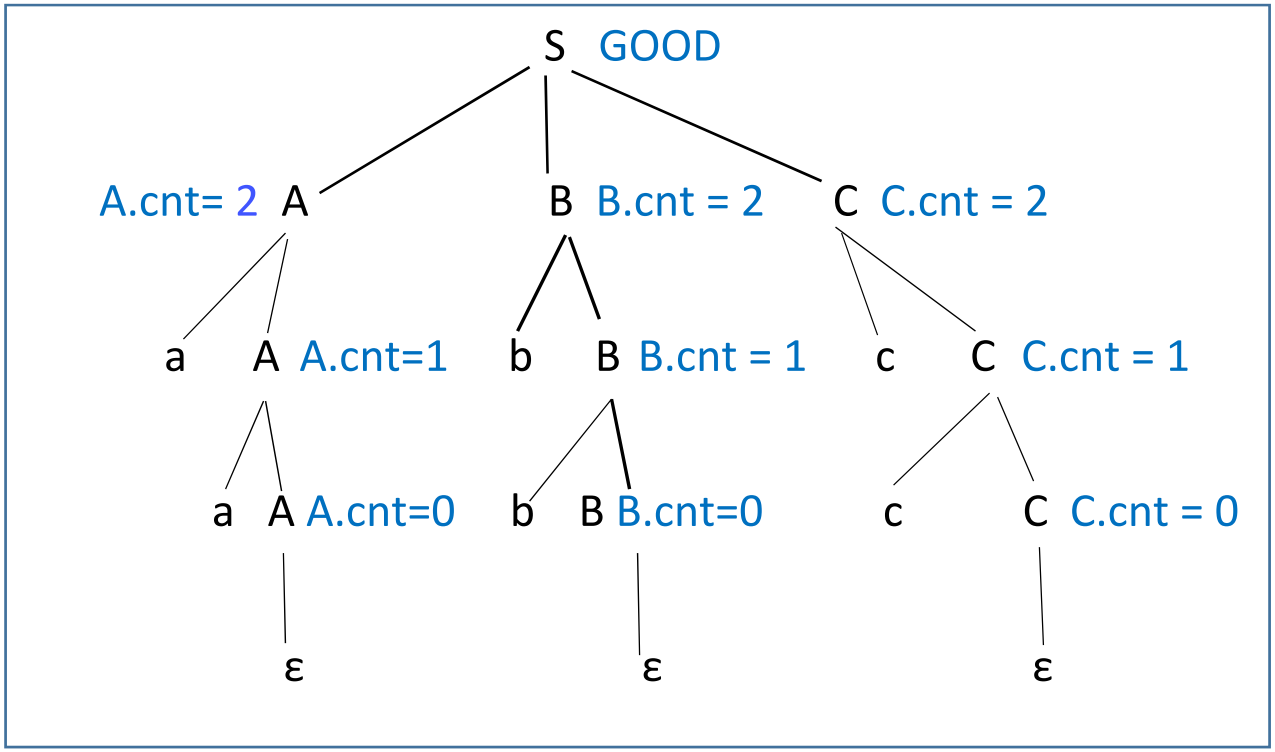
A → ε { A.cnt = 0 }



$A \rightarrow aA_1 \qquad \{ A.cnt = A_1.cnt + 1 \}$



$A \rightarrow aA_1 \qquad \{ A.cnt = A_1.cnt + 1 \}$



Attributed tree (עץ מורחב, עץ מקושט) –

result of the semantic analysis

Syntax-directed definition
הגדרה מונחית תחביר

<u>Its elements:</u>

- Grammar rules
- **Semantic attributes** of grammar symbols (in this example – counters).
- Attribute **represents a property/knowledge** needed for a problem solution.
- **Semantic actions**; each action is associated with a grammar rule.
  Actions are applied to semantic attributes, to calculate their values

<u>Why "syntax-directed" ?</u>

- Every time a grammar rule is applied, the associated action should be executed

Order in which grammar rules and actions are applied

In the example:

- derivation is done top-down
- values of counters are calculated bottom-up

For an occurrence of A in a derivation tree:

- value of A.cnt is calculated only **after derivation from this A is fully completed**
- e.g. when $A \rightarrow aA_1$ is applied:
  to obtain the required info about the whole (A.cnt),
  we first have to obtain the info about its parts: a and A_1 .

Such attributes are called **"synthesized attributes" (תכונות נוצרות)**

Order in which grammar rules and actions are applied

But not always all attributes are synthesized.

Sometimes the info is obtained in the opposite direction:

- attribute of a whole is known
- this helps to calculate the attributes of its parts

Such attributes are called **"inherited attributes" (תכונות נורשות)**

Example – grammar describing declarations in a programming language

D → T L

T → int

T → real

L → id

L → id , L

E.g. G allows to derive these declarations:

int a, b, c   real x, y

Task: a syntax-directed definition that allows for each declared id to know its type

Why such definition is needed at all? Isn't the grammar enough for that?

E.g. for real x, y - specified explicitly that the type is real! What else is needed?

BUT: in the rule D → T L the type is specified for the entire list L

We want to know the type of every separate element in L.

Need to inherit info about L to all id's in L.

Task: a syntax-directed definition that allows for each declared id to know its type

Semantic attributes for solution of the problem:

- T.type – what type (int or real) T represents?
  Synthesized attribute – value known only after derivation from T

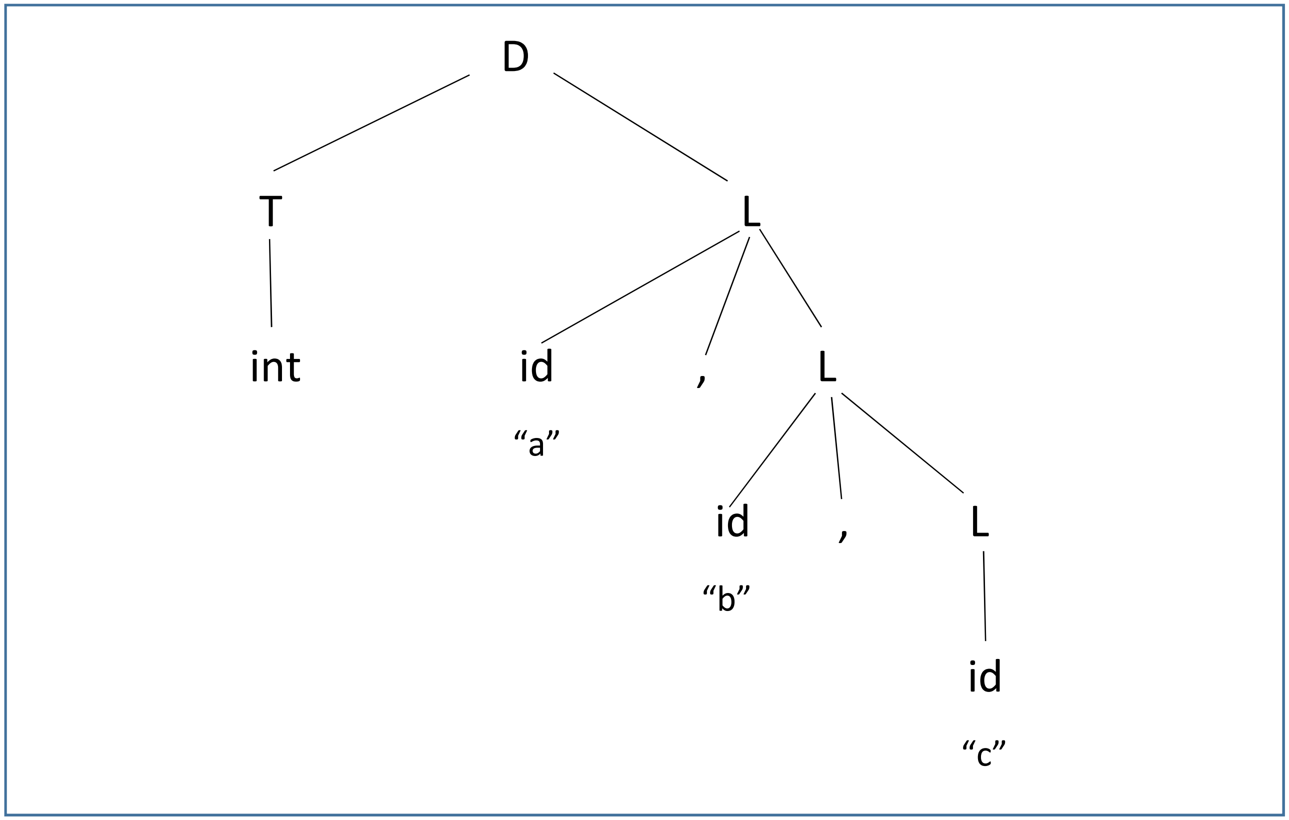
- L.type – type of the list L
  Inherited attribute - value known before derivation from L
  (in particular – before the amount and names of the id’s in L are known)

- id.type – id’s type

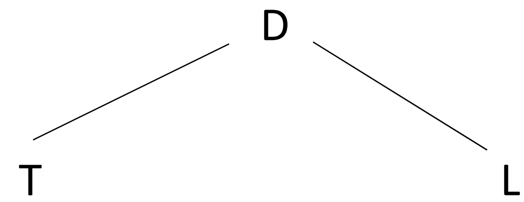
Solution - syntax-directed definition

$D \rightarrow T\ L$	{ L.type = T.type }	
$T \rightarrow$ int	{ T.type = int }	
$T \rightarrow$ real	{ T.type = real }	
$L \rightarrow$ id	{ id.type = L.type }	**inheritance from L**
$L \rightarrow$ id , L_1	{ id.type = L.type ; L_1.type = L.type }	**inheritance from L**

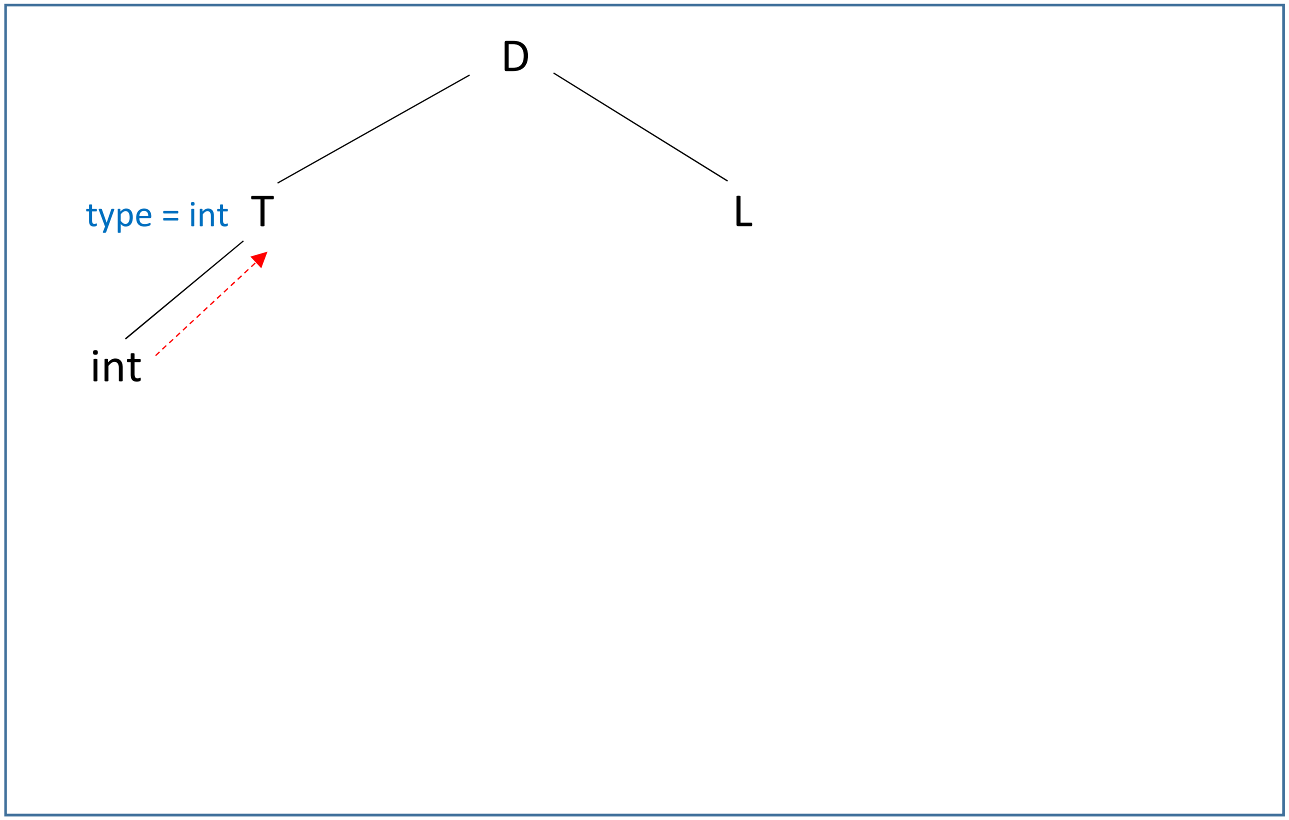
What is the order of derivation steps and semantic actions in this case?



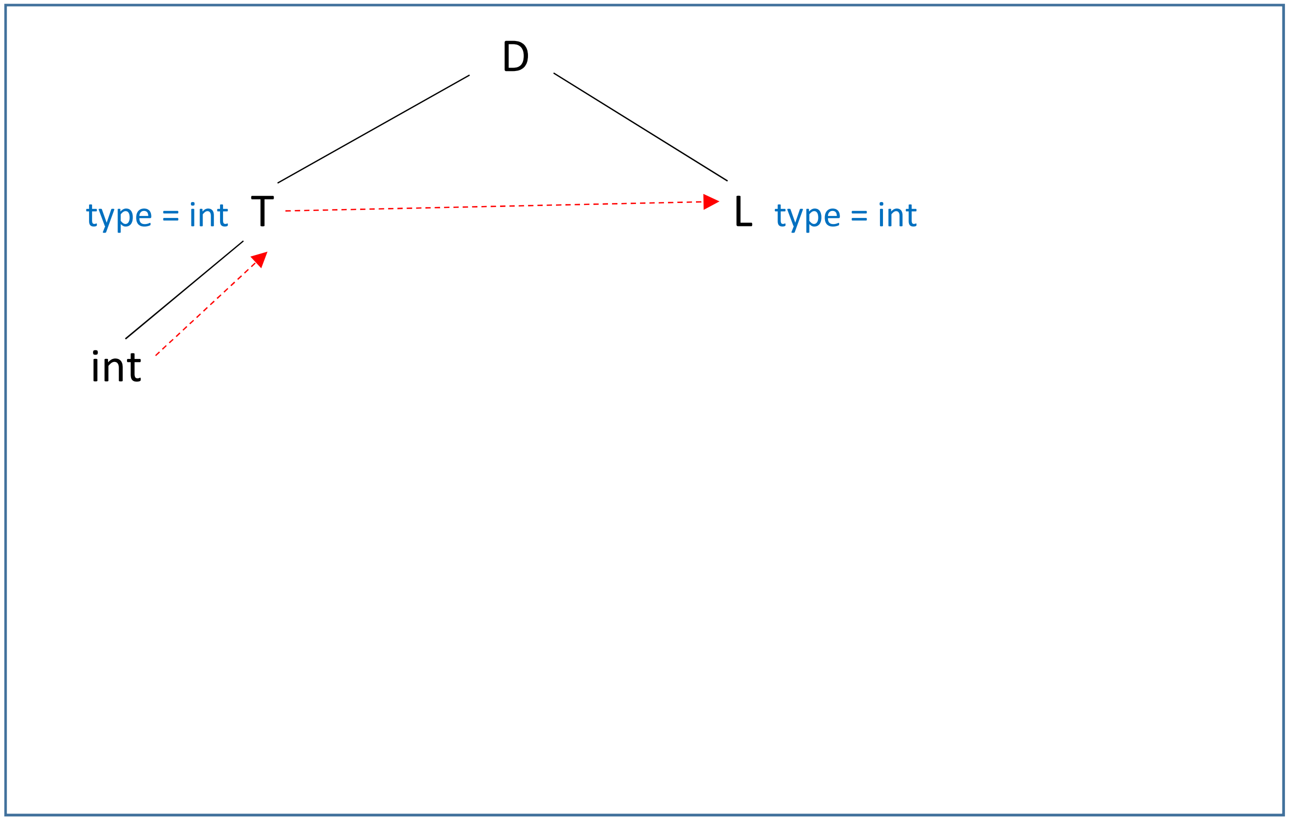
Derivation tree for int a, b, c



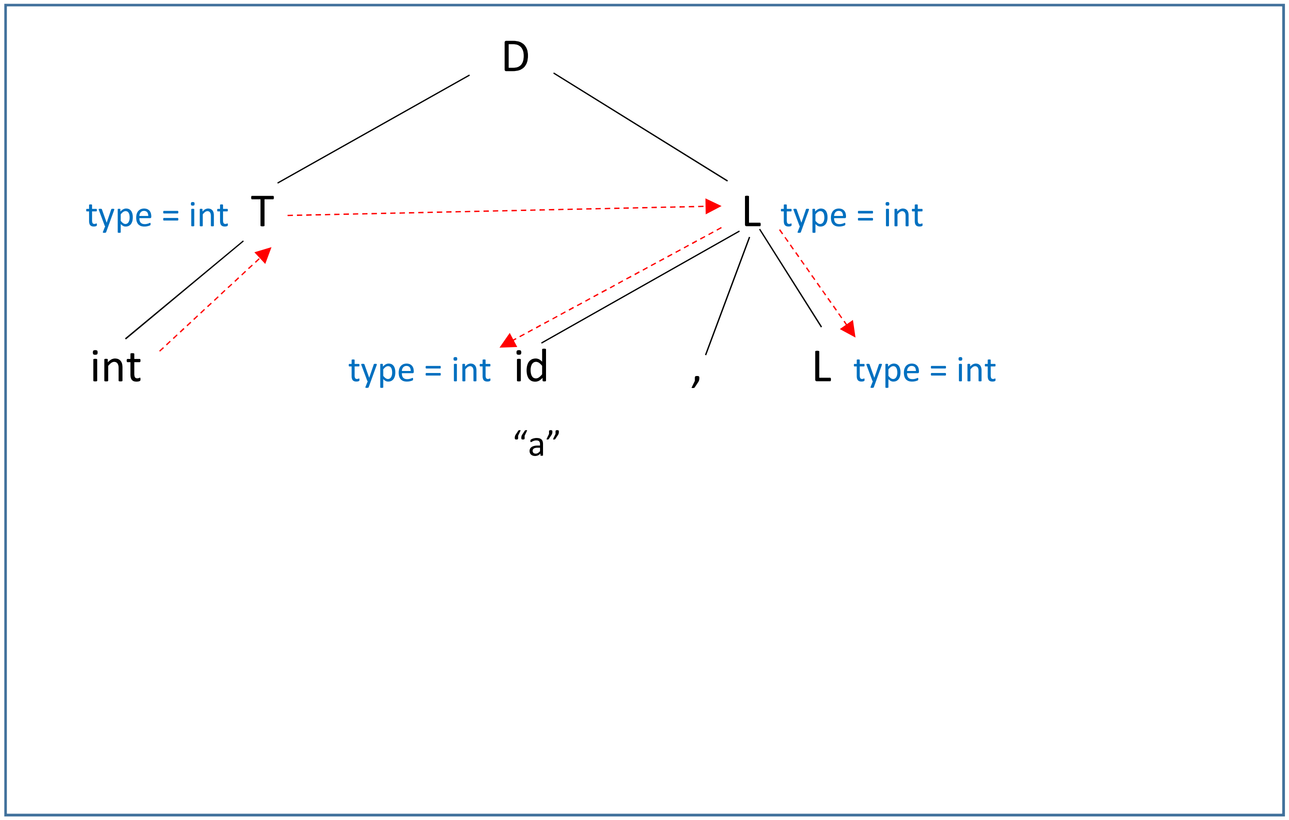
$D \rightarrow T \;\; L$



T → int { T.type = int }

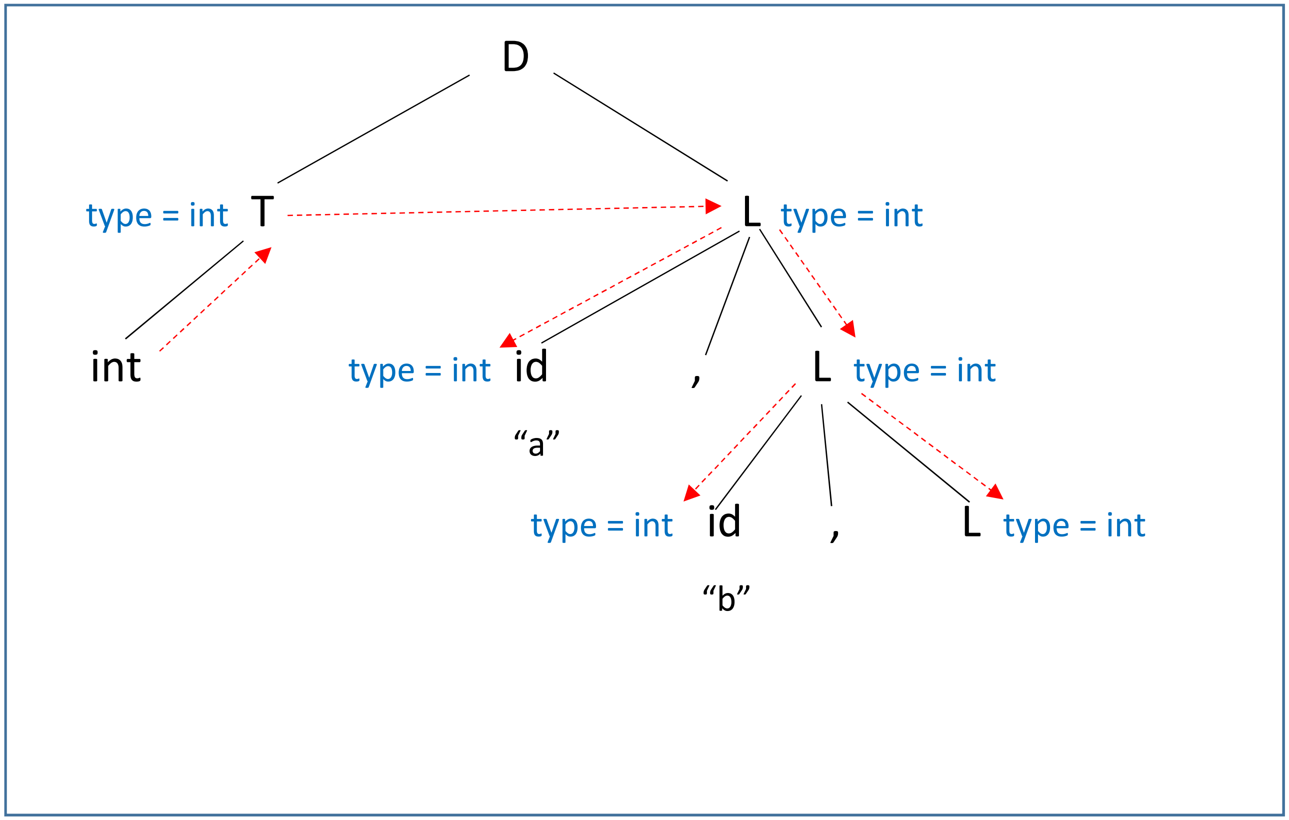


Inheritance from T to L



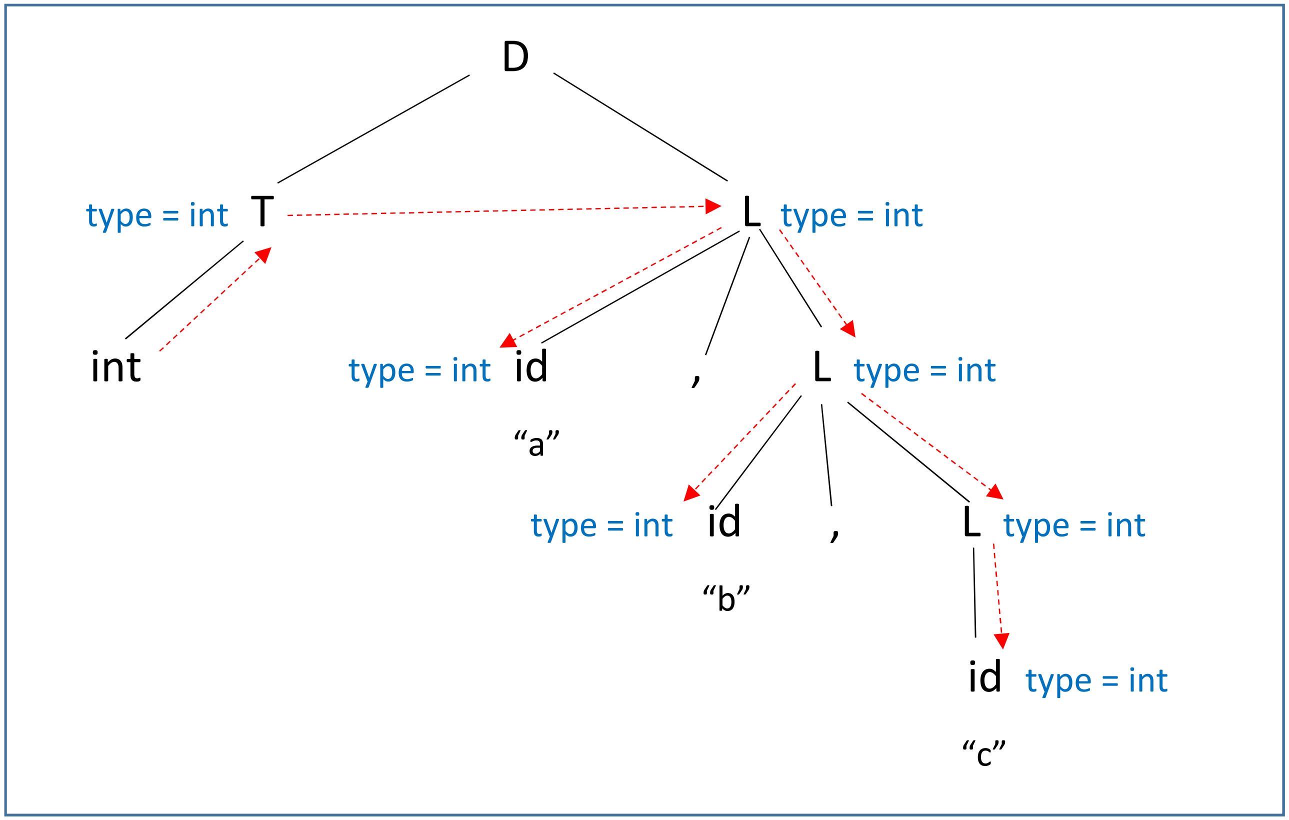
$L \rightarrow id\ ,\ L_1$ { id.type = L.type ; L_1.type = L.type }

/* inheritance from L */



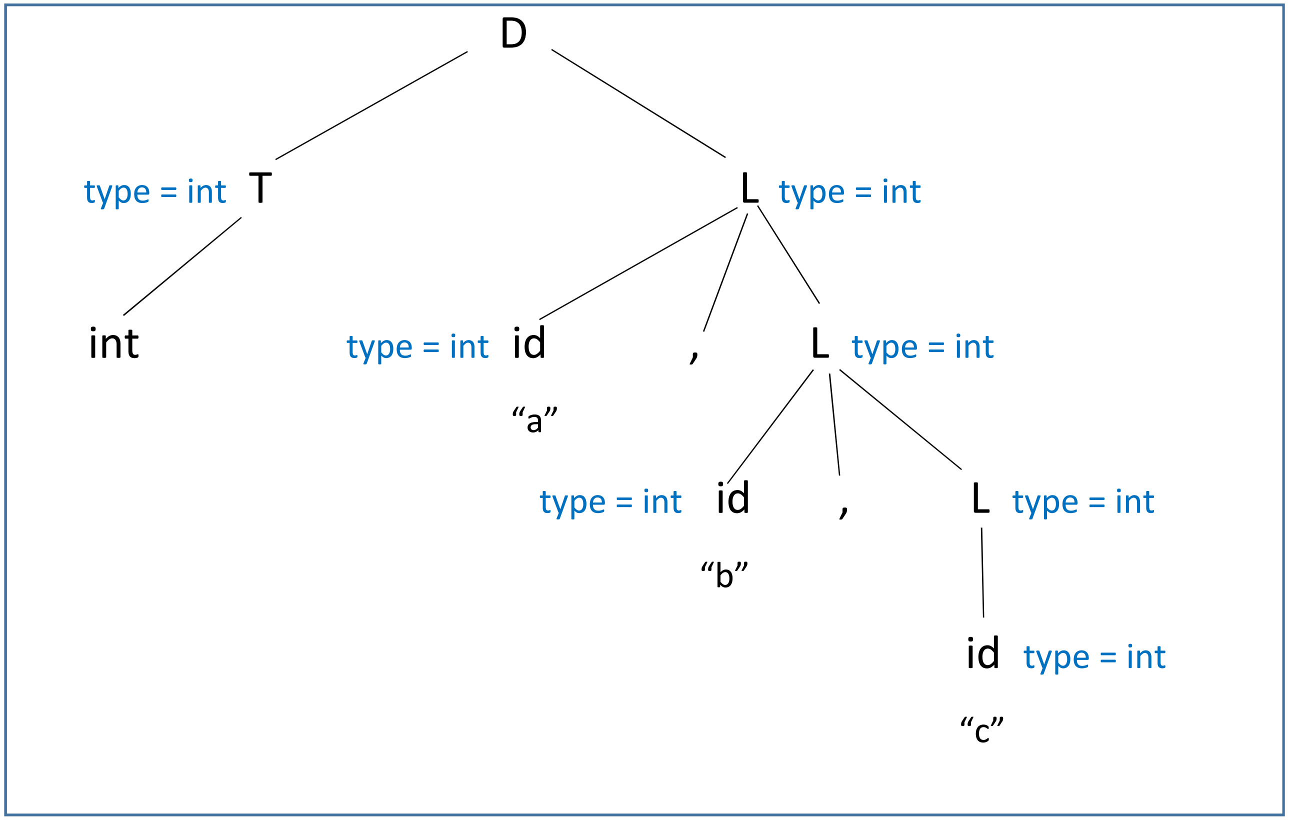
$L \rightarrow id\ ,\ L_1 \qquad \{\ id.type = L.type\ ;\ L_1.type = L.type\ \}$

/* inheritance from L */



L → id { id.type = L.type }

/* inheritance from L */



Attributed tree – the result of the semantic analysis

Solution – correct order of derivation steps and semantic actions

D → T { L.type = T.type } L

T → int { T.type = int }

T → real { T.type = real }

L → id { id.type = L.type }

L → id { id.type = L.type ; L_1.type = L.type } , L_1

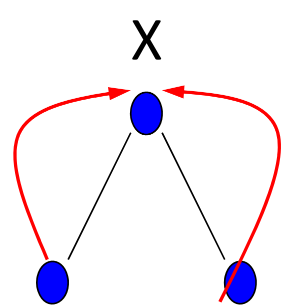
Summary - synthesized attributes (תכונות נוצרות)

Attributes that are passed in the parse tree **upwards**.

x

Value of a synthesized attribute of variable X at X-node :

- is computed from values of attributes in the children nodes
- i.e. **after completion of derivation from that occurrence of X**



Summary – inherited attributes (תכונות נורשות)

Attributes that are passed in the parse tree **downwards**.

Value of an inherited attribute of variable X at X-node :

- is computed from values of attributes in siblings and the parent of that node
- i.e. **before beginning of derivation from that occurrence of X**