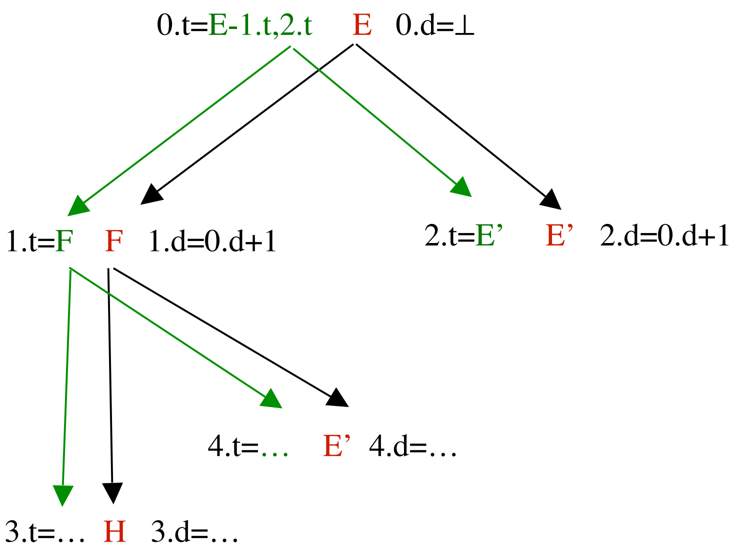
# Apply Attribute Grammars to Parse Trees: An Exercise

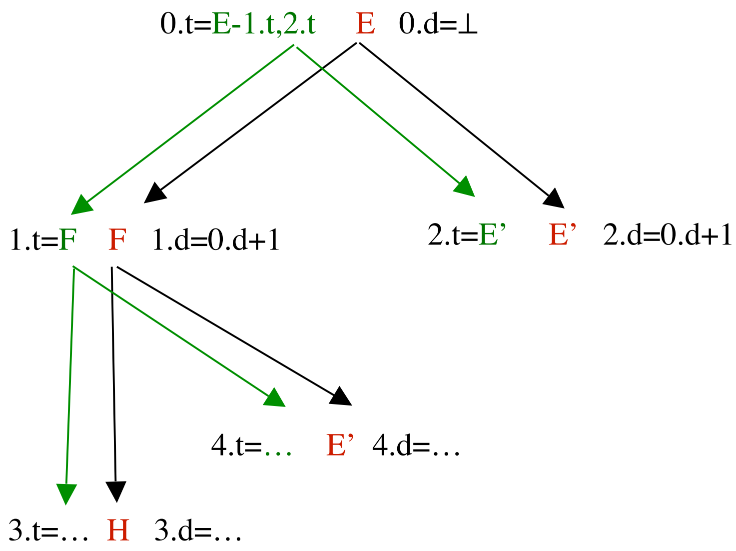
Let *mk-tree* be an arity-variant procedure that applies to a *label* and to n *trees* and results the tree rooted on a node having, as label, the first argument, and, as sons, the n trees, in the order of appearance from left. Say: 1) the type of the attributes; 2) the family $L_{\{ai\}}$; 3) the value of each attribute relating to 2+3.

E::FE' [0]  E'::=+E [1] | [2] ε
F::=HF' [3]  F'::=*F [4] | [5] ε
H::=num [6] | [7] (E)




| | |
|---|---|
| **E::= F E'** [0] | **E.tree:= mk-tree('E', F.tree, E'.tree), F.depth:=E.depth+1, E'.depth:=E.depth+1** |
| **E'::= + E** [1] | **E'.tree:= mk-tree('E'', mk-leaf('+'), E.tree), +.depth:=E'.depth+1,E.depth:=E'.depth+1** |
| **E'::= ε** [2] | **E'.tree:= mk-tree('E'', mk-leaf('ε')) ε.depth:=E'.depth+1** |
| **F::= H F'** [3] | **F.tree:= mk-tree('F', H.tree, F'.tree), H.depth:=F.depth+1, F'.depth:=F.depth+1** |
| **F'::= * F** [4] | **F'.tree:= mk-tree('F'', mk-leaf('*'), F.tree), *.depth:=F'.depth+1, F.depth:=F'.depth+1** |
| **F'::= ε** [5] | **F'.tree:= mk-tree('F'', mk-leaf('ε')) ε.depth:=F'.depth+1** |
| **H::= num** [6] | **H.tree:= mk-tree('H', mk-leaf(num)), num.depth:=H.depth+1** |
| **H::= (E)** [7] | **H.tree:= mk-tree('H', mk-leaf('('), E.tree, mk-leaf(')')), E.depth:= H.depth+1, (.depth:=H.depth+1, ).depth:=H.depth+1,** |

# Two Kinds of Attributes: Syntesized - Inherited




**Node 1 occurs, in the tree, in 2 different way:**

- **left side grammatical of**

  **E::= F E'**

- **right side grammatical of**

  **F::= H F'**

**Hence, attributes of node 1 can be defined in actions of:**

**2 different attribute productions:**

**This is the case of our grammar:**

**F.depth *is defined* in E::= F E'**

**F.tree *is defined* in F::= H F'**

**Attribute:**

**F.depth *is called* Inherited**

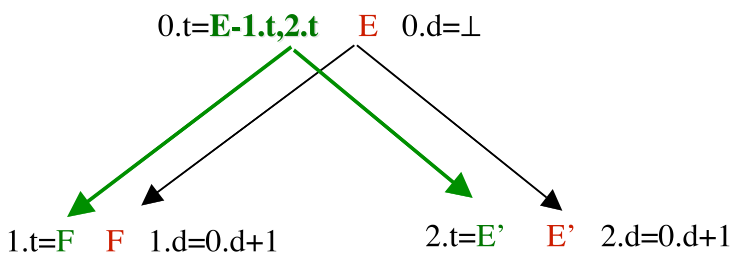
**F.tree *is called* Syntesized**

# Syntesized Attributes

Let $G^A \equiv \{\Sigma, V, s, P^A, \{a_i\}\}$ be an attribute grammar.
Let $p \equiv B ::= \beta\ \{\alpha\} \in P^A$.
Let X.a be an attribute occurring in $\{\alpha\}$. Then
X.a is a **synthesized attribute** if and only if one the two:

- $\exists\ X.a = e \in \{\alpha\}$ and $X \equiv B$
- $\exists\ X_i.a_{ij} = e_{ij} \in \{\alpha\}$ and $X.a \in Var(e_{ij})$ and $X \in Sym(\beta)$

where: $Sym(\beta)$ is the set of grammatical symbols in $\beta$
$Var(e)$ is the set of attributes occurring in e




**E::= F E’ { E.tree:= mk-tree('E', F.tree, E'.tree)…}**

**A Pragmatic View:**

- **Attribute of the node only depends from attributes of the sons**
- **It expresses Compositional Properties**

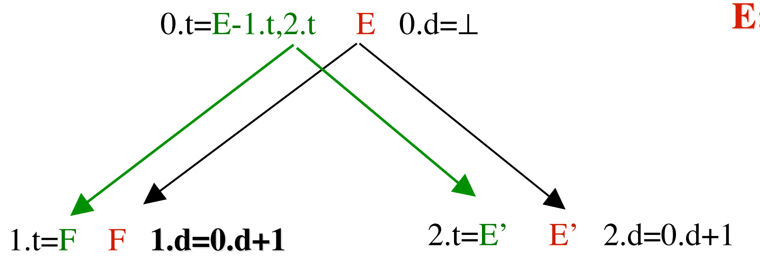
Let $G^A \equiv \{\Sigma, V, s, P^A, \{a_i\}\}$ be an attribute grammar.
Let X be a grammatical. Then
A-Syn(X) is the set of all Syntesized attribute of X in $G^A$

# Inherited Attributes

Let $G^A \equiv \{\Sigma, V, s, P^A, \{a_i\}\}$ be an attribute grammar.
Let $p \equiv B ::= \beta \ \{\alpha\} \in P^A$.
Let X.a be an attribute occurring in $\{\alpha\}$. Then
X.a is an **inherited attribute** if and only if one the two

- $\exists$ X.a=e $\in \{\alpha\}$ and X≡Sym(β)
- $\exists$ $X_i.a_{ij}=e_{ij} \in \{\alpha\}$ and X.a $\in Var(e_{ij})$ and X≡B

where: Sym(β) is the set of grammatical symbols in β
Var(e) is the set of attributes occurring in e




**E::= F E' {F.depth:=E.depth+1)…}**

**A Pragmatic View:**

- **Attribute of the node only depends from attributes of the context (father, brothers)**
- **It expresses Contextual Properties**

Let $G^A \equiv \{\Sigma, V, s, P^A, \{a_i\}\}$ be an attribute grammar.
Let X be a grammatical. Then
A-Inh(X) is the set of all Inherited attribute of X in $G^A$

# Applications of the Attribute Grammars

- Power: Context Sensitives and Attribute Grammars
- Attribute Evaluation: Three Execution Methods
- Oblivious and L-Attributed Grammars
- Bottom-up Executors for S-Attributed
- Top-down Executors for L-Attributed
- Bottom-up: Transformations for L-Attributed

# Attribute grammars are greatly powerful

because of the combination with a *meta* that can be a programming language

Consider the language $L_2$ on the right side. $L_2 \notin$ CF, and a Context Sensitive grammar for $L_2$ is shown.

**$L_2=\{u^n v^n z^n \mid n \geq 0\}$**

S::=A E
A::=u A v B | e
B v::= v B
B E::= z
B z::= z z

- **Such a grammar is difficult to write and even worse to analyze**
- **Context Sensitive Analyzers are complicated to build and impr-actical to use**
- **Attribute Grammars can be profitably used**

# Using an LL Attribute Grammar for Analyzing $u^n v^n z^n$

- Select a language $L_1 \in LL(1)$ including the language we are interested in:

  $u^n v^n z^n \subset L_1$

- Let G be a LL(1) grammar for $L_1$

  S' ::= S
  S ::= u S z | V
  V ::= v V | ε

- Extend G into an Attribute Grammar that computes an attribute of S' to true if and only if the analyzed string belongs to L(G), hence has form $u^n v^m z^k$ and n=m=k.

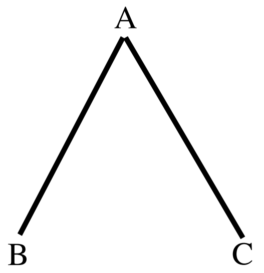
  S' ::= S **{S.r=(S.u==S.v)&(S.u==S.z)}**
  $S_1$ ::= u $S_2$ z **{$S_1$.u=S.u+1; $S_1$.v=$S_2$.v; $S_1$.z=$S_2$.z+1}**
  S ::= V **{S.u=0; S.v=V.v; S.z=0}**
  $V_1$ ::= v $V_2$ **{$V_1$.v=$V_2$.v+1}**
  V ::= ε **{V.v=0}**

# Attribute Evaluation

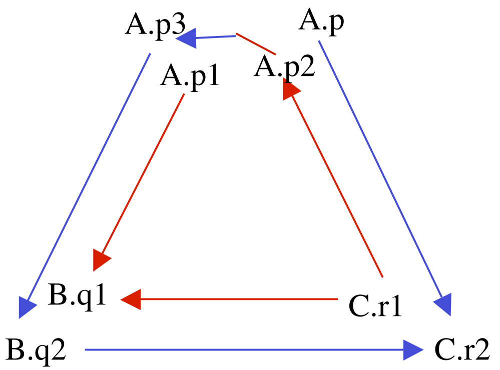
## Three different evaluation tools

1) **Parse Tree**:

- Construct the **Parse Tree**, T
- Construct the **Dependency Graph**, $T_d$ of T
- Find, if any, a **Topological Sort** $M_{Td}$ for $T_d$
- Visit $T_d$ according to $M_{Td}$ and Execute the actions associate to the nodes




**T**




**$T_d$**

**A.p, A.p1, C.r1**
**B.q1, A.p2**
**A.p3,**
**B.q2**
**C.r2**

**$M_{Td}$**

| Node | ActionAttribute |
|---|---|
| C | C.r1 |
| A | A.p2 - A.p - A.p1 |
| A | A.p3 |
| B | B.q1 - B.q2 |
| C | C.r2 |

**Visit**

**-Only for Multi-Pass Parser/Compiler**
**-Method applies at Compile Time**

# Attribute Evaluation

## Three different evaluation tools - 2

2) **Rule-based**:

- For each production:
  - Analyze the meaning of the actions occurring in it
  - State a **proper execution order** for the actions
- Combine such an order with the Parse-Tree constructor:
  - Only one Code for Parse-Tree construction and Action execution
  - **Versus** Distinct, Correlated, Codes

**Ad Hoc Construction: The resulting code is hard to modify**

**- Also for one-Pass Parser/Compiler**
**- Method applies at Compile Construction Time**

# Attribute Evaluation

## Three different evaluation tools - 3

3) **Oblivious**:

- The **execution order** for the actions is established according to:
  - **same criteria** for all propdductions
  - criteria that **ignore the meaning** of the actions
  - but are adequate for **executing actions in the correct way**
- Action Execution is combined with Parsing:
  - **Top-Down Executors**
  - **Bottom-Up Executors**
- **Parser Generators** are extended to **Attribute Grammar Evaluators**

**- Only for one-Pass Parser/Compiler**
**- Method applies at Compile Constrution Time**

# Attribute Evaluation

## **Parser Generators** as **Attribute Grammar Oblivious Evaluators**

- **How can Parsing and Action Evaluation be combined ?**
  - At each derivation/reduction, the production actions are evaluated

- **When actions are evaluated in this way, what part of the Parse-Tree has already been traversed and then, known to the actions?**
  - The nodes of a **Depth-First** visit of the Parse-Tree up to the current input:
    - + **Top-Down: Preorder Depth First**
    - + **Bottom-up: Postorder Depth First**

- Parser Generators can be extended into Oblivious Evaluators of a attribute grammar G if:

**Depth-First visit is a Topological Sort of the Dependency Graph of G**

# L-Attributed Grammars

L-Attributed Grammars is **a class** of Attributed Grammars (or SDD) that has **Depth-First** as a **Topological Sort** of the **Dependency Graph** of the **Parse-Tree attributes** of the grammar.

Let $G^A \equiv \{\Sigma, V, s, P^A, \{a_i\}\}$ be an attribute grammar.
Let $p \equiv B := B_1 \ldots B_n\{\alpha\} \in P^A$.
$G^A$ is L-attributed if and only if:
$\forall X_i.a_{ij} = e_{ij} \in \{\alpha\}$ for $X_i \in Sym(B_1 \ldots B_n)$:
if $X_k.a_{ik} \in Var(e_{ij})$ then:
- either $X_i \equiv B_{hi}$, $X_k \equiv B_{hk}$ and $1 \le h_k \le h_i \le n$
- or $X_k \equiv B$ and $a_{ik} \in A\text{-}Inh(B)$

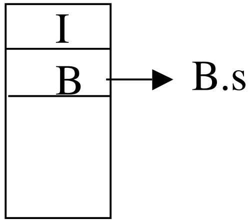
- **S-attributed Grammars** are containing only synthesized attributes
- S-attributed are L-attributed.

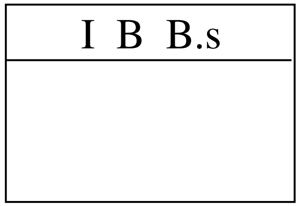
Theorem. If G has Top-Down/Bottom-up Parser and $G^A$ is L-attribued then **$G^A$ has Top-Down/Bottom-up oblivious evaluator**

# Bottom-Up Evaluator for S-attributed
# How do it by extending LR Parsers

Extend the values of the push-down automata, LR control stack:

- Associate to each grammatical symbol B:
  - the syntesized attributes or none (if it has no attribute)
  - the transtion state of LR analysis




- At each reduction with handle A::=B1…Bn {$\alpha$} compute all the actions in {$\alpha$}.
 - Let $A.a_i=e_i$ be one of them.
   If $e_i$ contains occurrences of attributes of the grammatical $B_i$ then:
   - access (n-i)-th position, below the top of the stack, and
   - select the value $I_i$ $B_i$ $[v_i]$ (where $[v_i]\equiv v_{i1}\dots v_{in}$) and find the correct $v_{ij}$
 - Let $[v]\equiv v_1\dots v_m$ be the values resulting for the attributes $a_1\dots a_m$ of A.
- Reduce and insert $I_j$ A [v], where $I_j$ is the transition state of LR analysis.

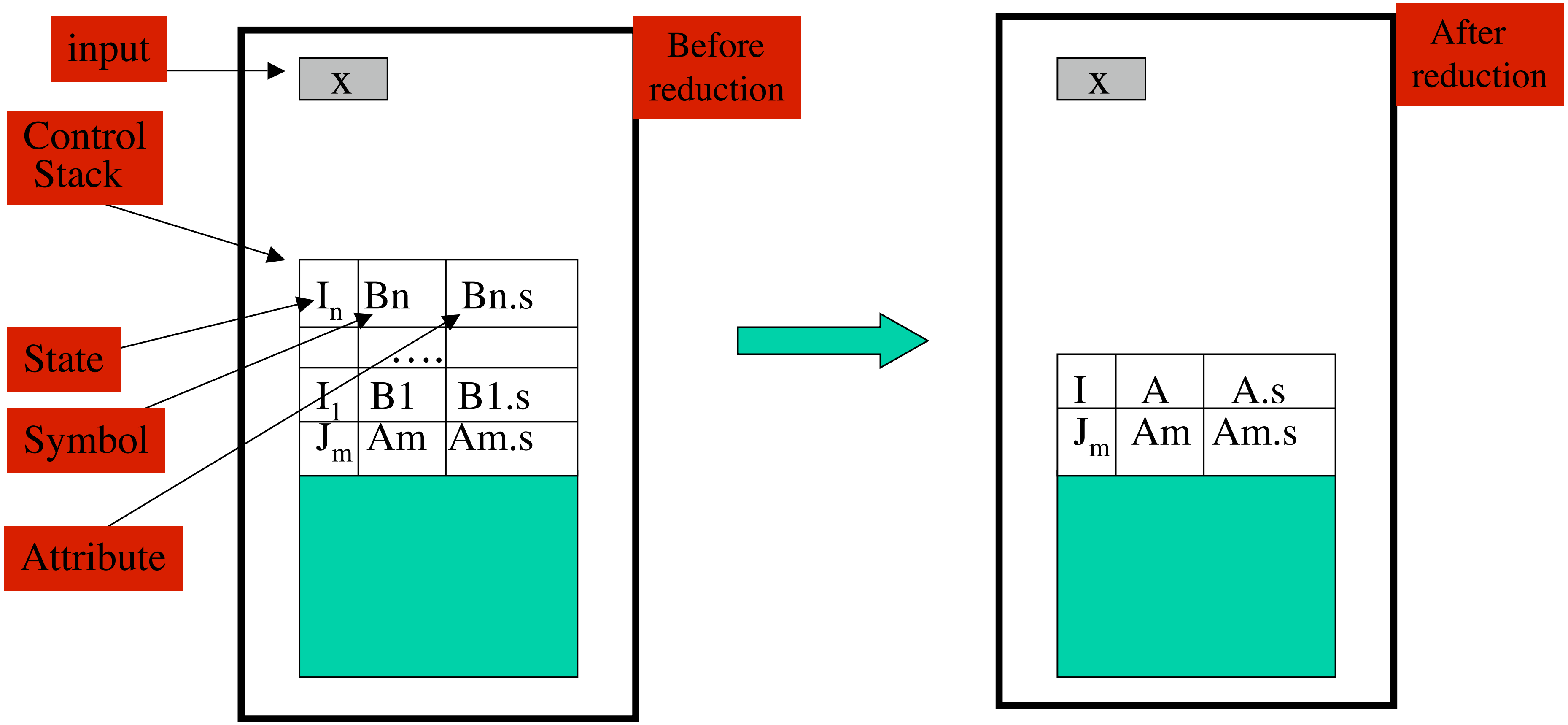
# How do it: LR Control Stack

**production**

(k) A::=B1…Bn {α}

**LR Table**

$Action(I_n,x)=R/k$ $Goto(J_m,A)=I$




Each B*i* and its attributes B*i*.s are computed by the previvious reductions (sons - depth first)

A.s=α has been just computed: It can only depend from A-Syn(B*i)* (A's sons) that are in the stack

# Top-Down Evaluators for L-Attributed
# From L-Attributed to Translation Schemes

**Translation Schemes** = Grammars with Productions where actions and grammatical symbols are mixed

$$A::=\{\beta 1\}B1...\{\beta k\}Bk\{\alpha\}$$

in a way that:

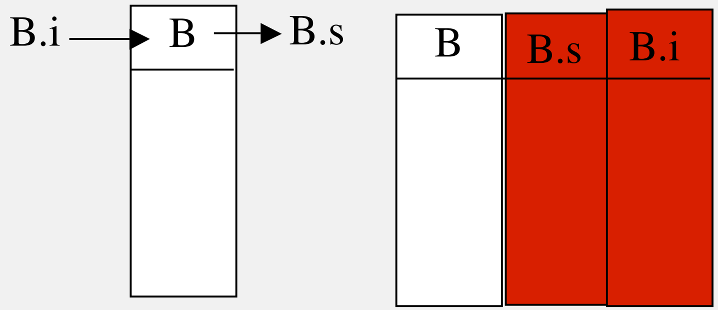
- A-Inh(Bi) are defined only in actions {βi} that precede Bi (for ach i)
- A-Syn(A) are defined in {α}

If G is L-attributed, its TS has actions that can use only, attributes of symbols that precede the actions.

# Top-Down Evaluator for L-attributed
# How do it by extending LL Parsers

- **Transform** L-attributed **in Translation Scheme**
- **Pair** the LL control stack, C, with
  - one data **stack for synthesized** values, **S**,
  - one data **stack for inherited** values, **I**.
- **Extend C to contain actions**:
  - **At each derivation** with **A::={β1}B1...{βk}Bk{α},**
    - {β1}B1...{βk}Bk{α}
  - (Let B0≡A and $\beta_{k+1}$≡α)

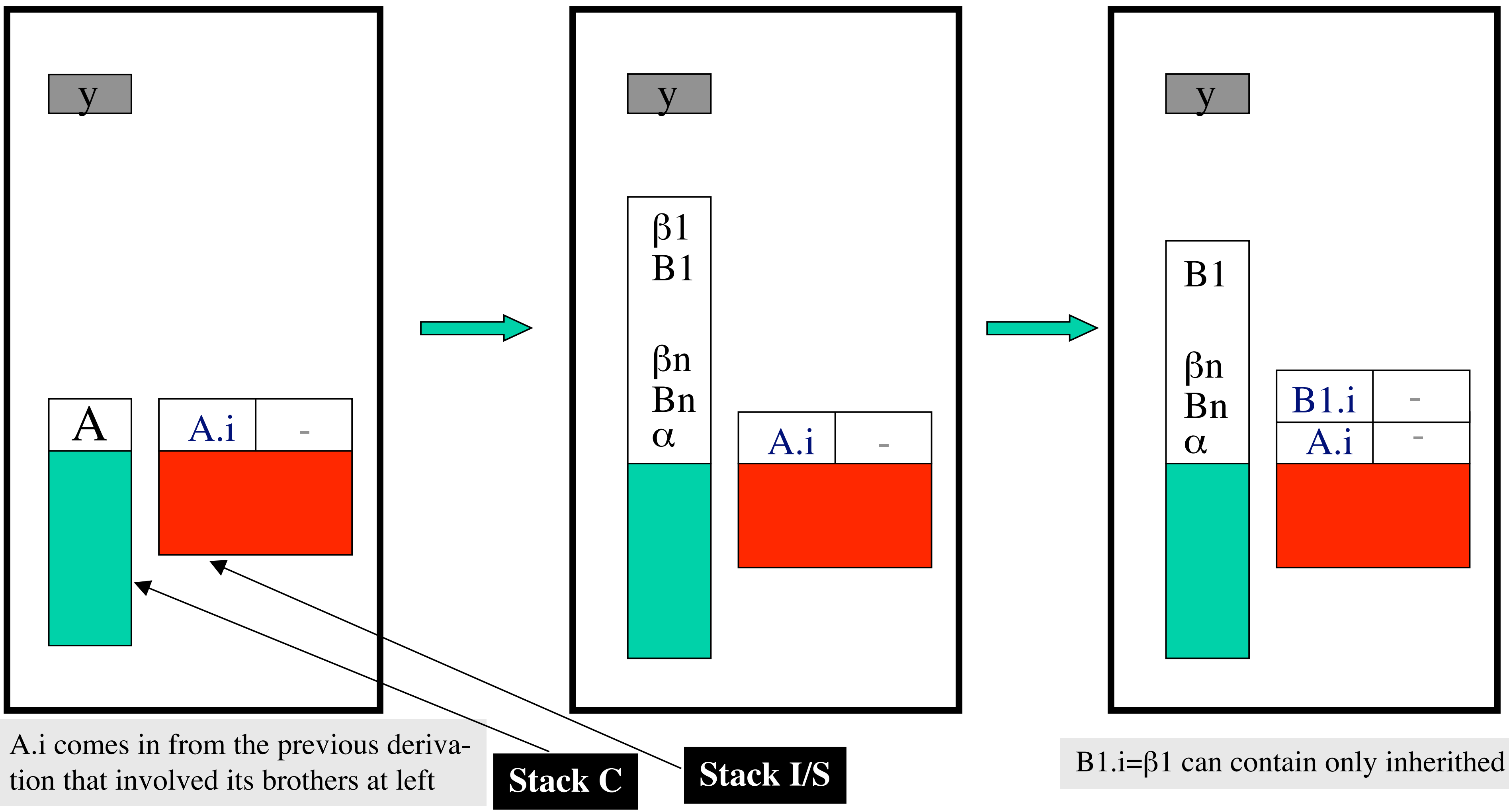
    **When an action βi** (1≤i≤k+1) is selected from the top of C
    - Action **is evaluated:**
      - **by using the evaluator** of Meta, and
      - **by replacing attributes** of:
        - **Bj** (j<i) with the values extracted, from I or S, at the (i-j-1)-th position from top
        - **A -** as above, by letting: B0≡A and $\beta_{k+1}$≡α
      - **by putting its result on**:
        - the **top of I**, if action is βi
        - **k-th position below** top of S, if action is α



# How do it: LL Control Stack - 1

(k) A::={β1}B1…{βn} Bn {α}

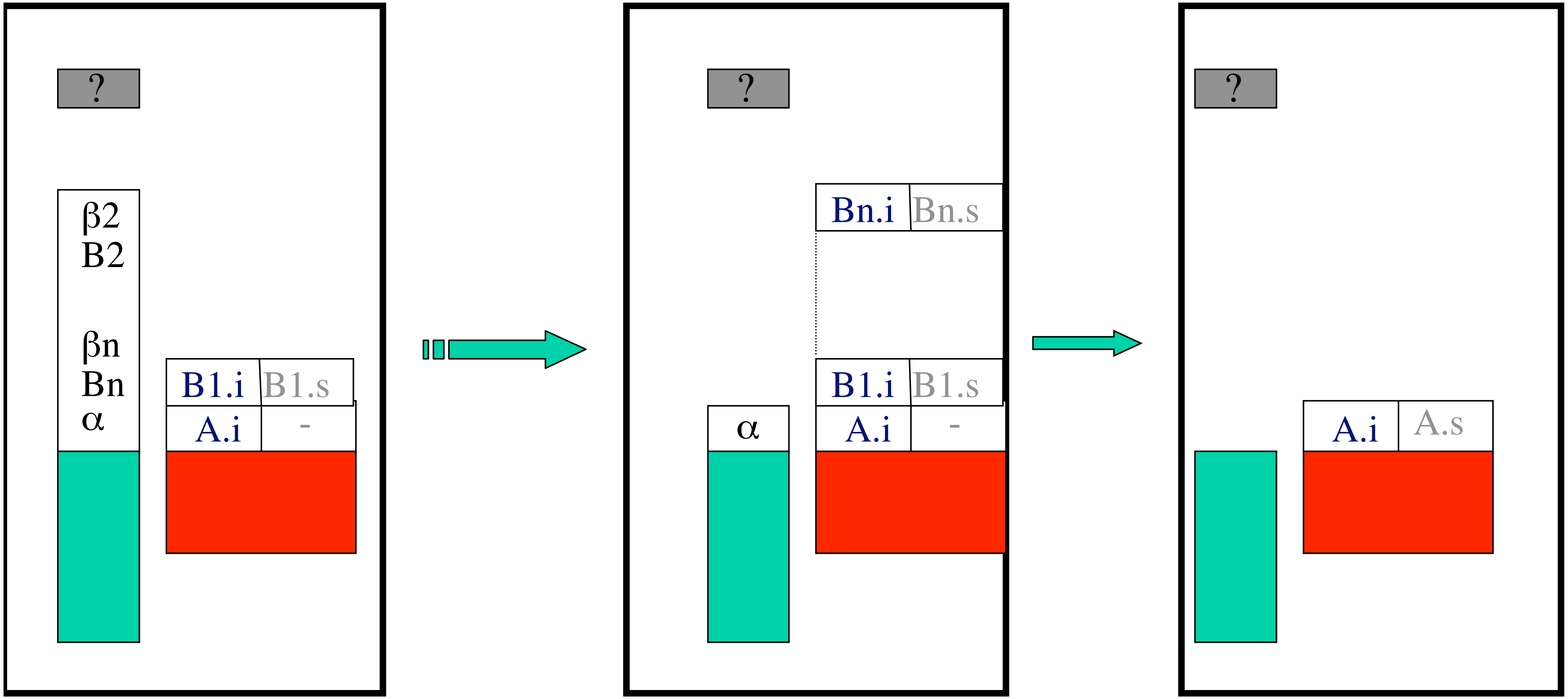
M(A,y)=k




A.i comes in from the previous derivation that involved its brothers at left

B1.i=β1 can contain only inherithed of A

# How do it: LL Control Stack - 2




All the attributes that α can use

Top of Data Stacks just after the derivation from A completes

# L-attributed Bottom-up Transformations: Markers

**Translation Descendant Scheme**

Transform: (n) $A::=\{\beta 1\}B1...\{\beta k\}Bk\{\alpha\}$

in

$A::=M_{n1}\ B1...\ M_{nk}\ Bk\{[\alpha]\}$

$M_{n1}::=\varepsilon\ \{[\beta 1]\}$

...

$M_{nk}::=\varepsilon\ \{[\beta k]\}$

**Translation Ascendant Scheme**

**Inner Actions** of the descendant schemes are transformed into final actions of ε-productions that are introduced by the Markers.

One **Marker uniquely identifies the position,** inside a production, and allows to handle:
*inherited attributes* of a symbol as *synthesized attributes* of a marker

# L-attributed Bottom-up Transformations: Factorization

Descendant Scheme

Ascendant Scheme

Transform: (n) $A ::= \{\beta 1\} B1 \ldots \{\beta k\} Bk \{\alpha\}$

in

$A ::= A_{nk}\ Bk\{[\alpha]\}$

$A_{nk} ::= A_{nk-1} Bk\text{-}1\ \{[\beta k]\}$

…

$A_{n2} ::= A_{n1}\ B1\{[\beta 2]\}$

$A_{n1} ::= \varepsilon\ \{[\beta 1]\}$

**Inner Actions** of the descendant schemes are transformed into final actions of productions of the new added grammaticals that are as many as the positions inside the production.

The new symbol Anj **uniquely identifies the j-th position,** inside n-th production of A, and allows to handle:

*inherited attributes* of a symbol as *synthesized attributes* of new symbol

# Marker Based Transformation
# How do actions have to be changed?

S::=U {U.n=0;}
$U::=U_1$ a {$U_1$.n=U.n+1;}
U::=ε {print(U.n);}

Translation scheme

S::= {U.n=0;} U
U::= {$U_1$.n=U.n+1;} $U_1$ a
U::= {print(U.n);}ε

MARKERS

S::= M1 U
U::= M2 $U_1$ a
U::= M3 ε
M1::= ε{U.n=0;}
M2::= ε{$U_1$.n=U.n+1;}
M3::= ε{print(U.n);}

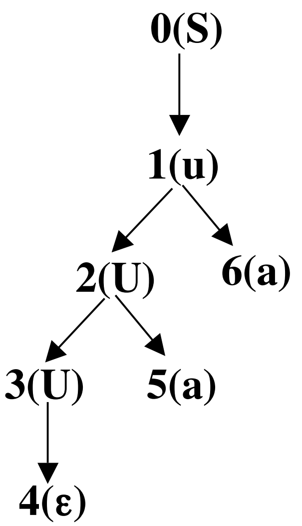
Stack Action Translation

S::= M1 U
U::= M2 $U_1$ a
U::= M3 ε
M1::= ε{push 0;}
M2::= ε{push(top+1);}
M3::= ε{print(top);}

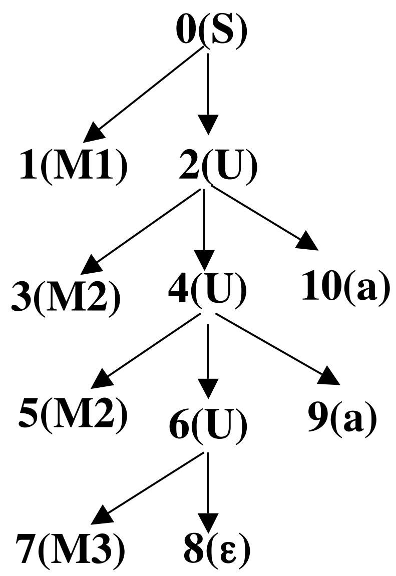
# Marker Based Transformation
# How do Parse Trees change?

| | |
|---|---|
| S::=U | {U.n=0;} |
| U::=$U_1$ a | {$U_1$.n=U.n+1;} |
| U::=ε | {print(U.n);} |

S::= M1 U
U::= M2 $U_1$ a
U::= M3 ε
M1::= ε{push 0;}
M2::= ε{push(top+1);}
M3::= ε{print(top);}




**How does depth-first tree visit change (postorder)**




**4->3->5->2->6->1->0**

**1->3->5->7->8->6->9->4->10->2->0**

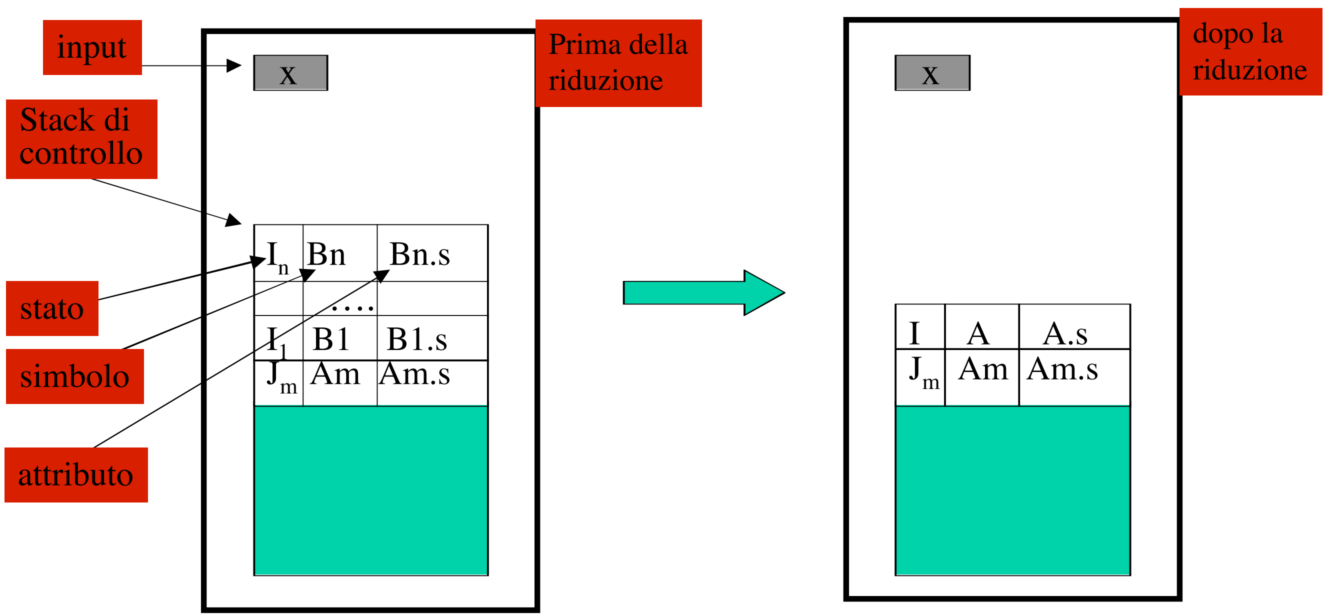
# Marker Based Transformation
# Attribute Evaluation -1

**produzione**

(k) A::=B1…Bn {α}

**Tabelle riconoscitore LR**

Action($I_n$,x)=R/k  Goto($J_m$,A)=I

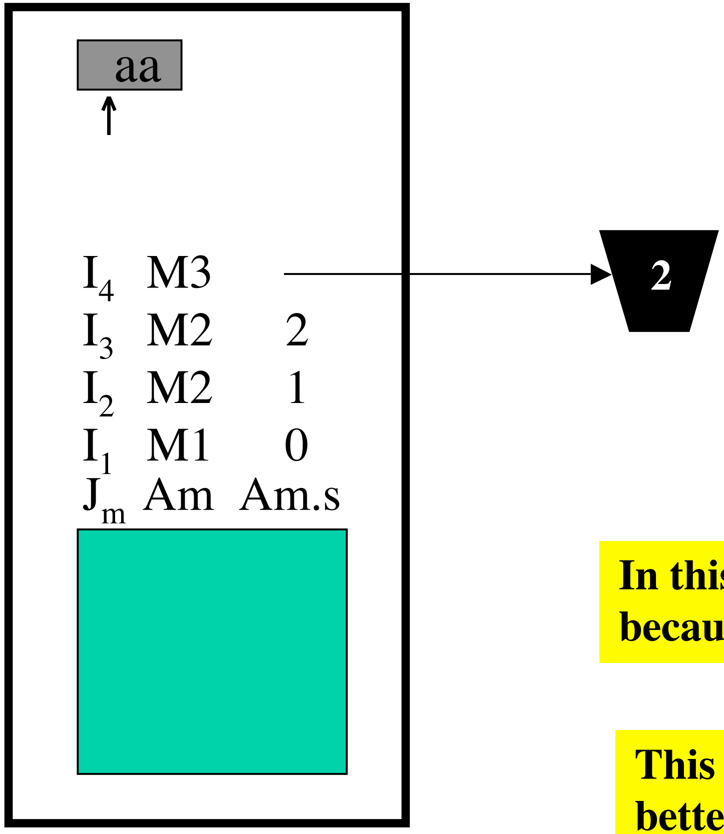



S::= M1 U
U::= M2 $U_1$ a
U::= M3 ε
M1::= ε{push 0;}
M2::= ε{push(top+1);}
M3::= ε{print(top);}

B*i* e i loro attributi B*i*.s sono stati calcolati dalle preceden-ti riduzioni (figli - depth first)

A.s=[α] e' calcolato: puo' dipendere solo da sintetizzati di B*i* (figli di A) tutti sullo stack

# Marker Based Transformation Attribute Evaluation -2




S::= M1 U
U::= M2 $U_1$ a
U::= M3 ε
M1::= ε{push 0;}
M2::= ε{push(top+1);}
M3::= ε{print(top);}

**In this case, evaluation cannot behave in this way because the new grammar is not more LR(1)**

**This is why Factorization may be considered a better alternative to the use of Markers.**

# Oblivious Evaluators Implementation

- **Top-down**:
  - Translation Invariants
  - Translation of actions, $\alpha$, containing attributes in actions on I/S stack positions
- **Bottom-up**:
  - Translation Invariants
  - Translation of actions, $\alpha$, containing attributes in actions on C stack positions

**matters not covered**

Analisi Statica