Grammatiche ad attributi

Una metodologia,

una tecnica,

uno strumento per

analisi (anche contestuali) e

generazione di codice

PARSING

Riconoscitori per

appartenenza: legalità della frase (o stringa)

struttura: decomposizione della frase in sotto-frasi
(parse tree)

Il linguaggio L definito, in modo ambiguo, dalla grammatica sotto.

E::= E (+|*) E | num | (E)

num * (num + num)

E::= F + E | F
F::= H * F | H
H::= num | (E)

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

	+	*	num	(	)	$
E			0	0		
E'	1				2	2
F			3	3		
F'	5	4			5	5
H			6	7		

E::= E (+|*) E | num

num * (num + num)

E::= T A | T
A::= (+|*) E
T::= num | “(“ E “)”

Anche questa grammatica genera il linguaggio L. Ma le frasi riconosciute hanno struttura diversa.

Le frasi di un linguaggio sono:

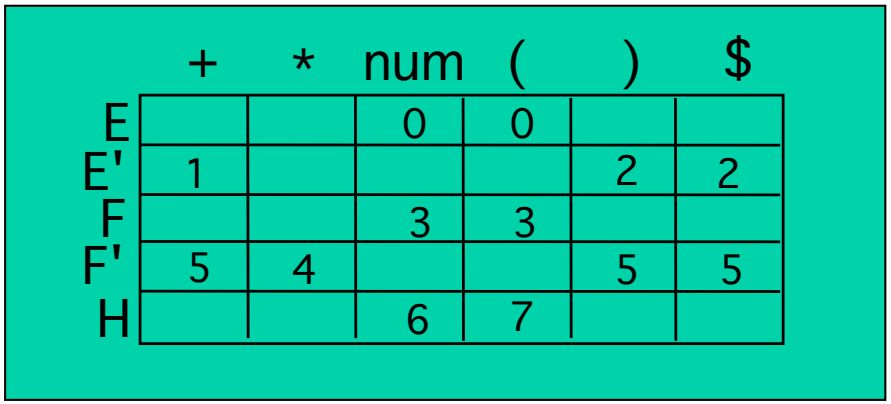
- **sequenze di simboli, *oppure***
- **le strutture con le quali sono provate appartenere alla grammatica che definisce il linguaggio?**

Dai riconoscitori ai generatori

- Parse tree
- Syntax e abstract tree
- Estendiamo i riconoscitori
- Attributi per i simboli grammaticali

I riconoscitori perdono la struttura

num * (num + num)



	+	*	num	(	)	$
E			0	0		
E'	1				2	2
F			3	3		
F'	5	4			5	5
H			6	7		

E -> FE' -> HF'E' -> numF'E' -> num*FE' -> num*HF'E' ->

-> num*(E)F'E' -> num*(FE')F'E' -> num*(HF'E')F'E' ->

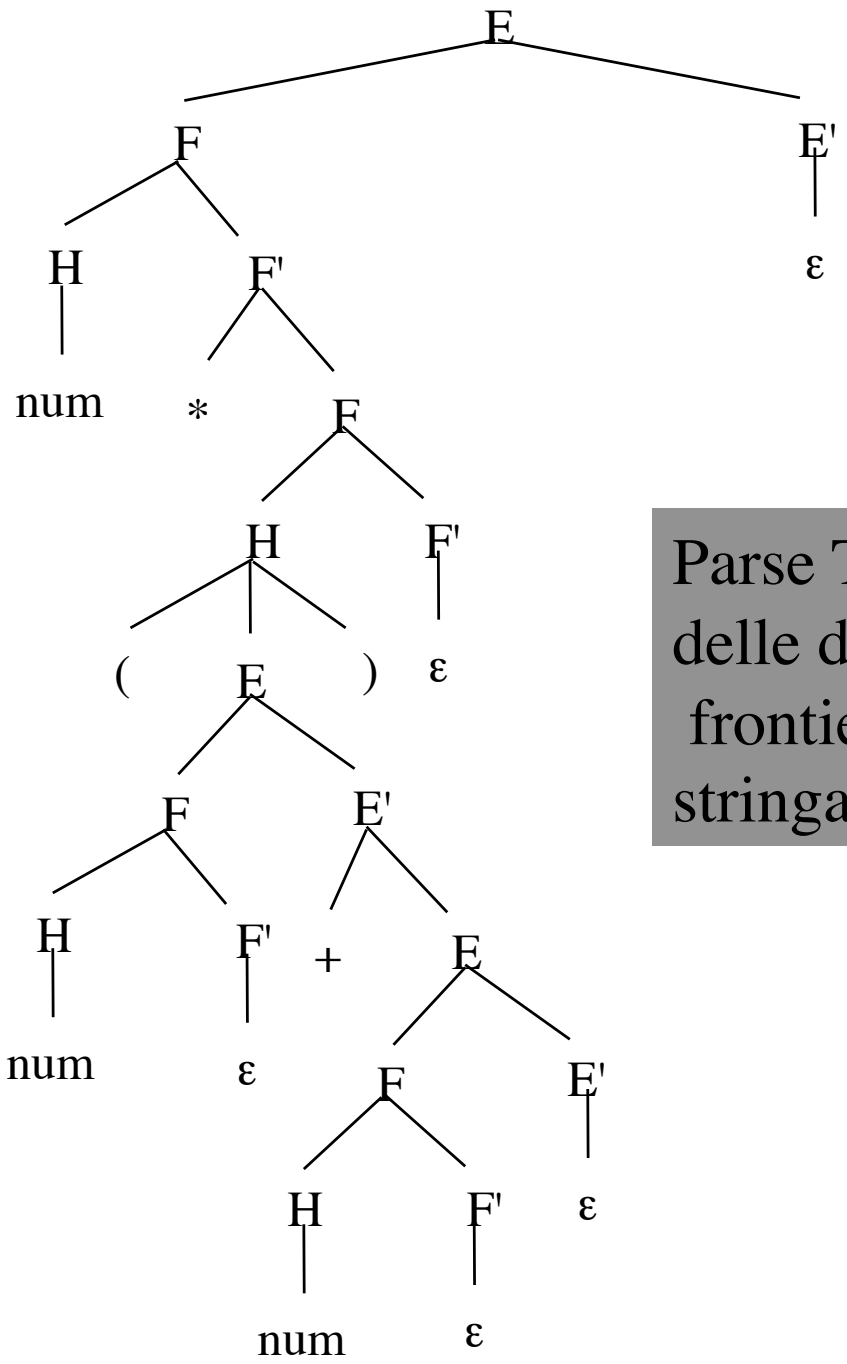
-> num*(numF'E')F'E' -> num*(numE')F'E' -> num*(num+E)F'E'

-> num*(num+FE')F'E' -> num*(num+HF'E')F'E'

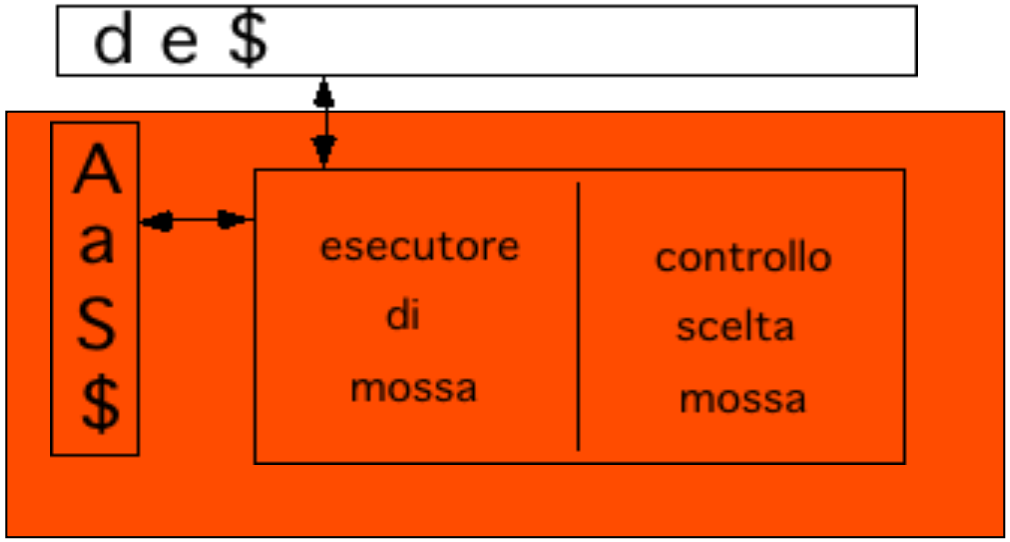
-> num*(num+numF'E')F'E' -> num*(num+numE')F'E'

-> num*(num+num)F'E' -> num*(num+num)E'

-> num*(num+num)



Parse Tree: mostra la struttura delle derivazioni con cui la frontiera e' riconosciuta come stringa appartenente alla grammatica



cambiamo le mosse:

derivare

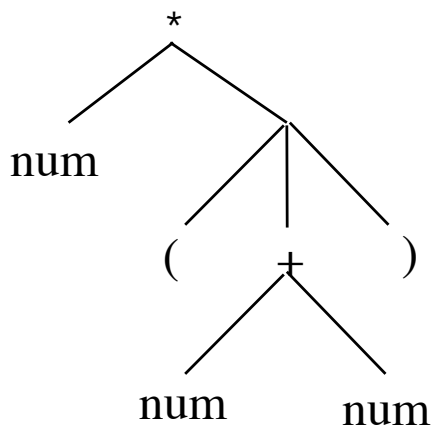
corrisponde a

creare una struttura di albero

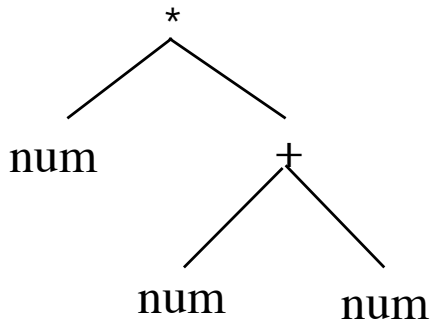
E -> F E'

corrisponde

E->F E' {tree(E):=mk-tree('E',tree(F),tree(E'))}



Syntax tree: contiene tutti i terminali inclusi quelli non **significativi**



Abstract tree: contiene i soli terminali **significativi**

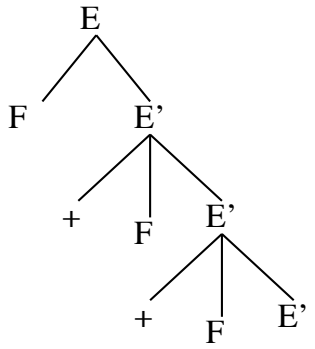
Relazione binaria: albero, albero etichettato

Alberi (e relazioni binarie):

- $T = \langle R, E \subseteq R^2 \rangle$
- $T = \langle R, E \subseteq R^2, L:R \to U \rangle$ -- *etichettati su U*

Dove: R: *insieme nodi* E: *insieme archi (orientati)* L: *funzione etichette su U*

Operazioni: root: $T \to R$, arity: $R \to N$, son: $T \times N \to T$
Selettori: $R(T)= R$, $E(T)=E$, $L(T)=L$
Abbreviazioni: sons: $T \to T^N$, L: $T^N \to U^N$
$sons(T)=son(T,1)\ldots son(T,arity(root(T)))$ $L(T_1,\ldots,T_k)=L(root(T_1)\ldots L(root(T_k)$



R(T) = {0,1,2,3,4,5,6,7,8}
L(T) = {(0,E),(1,F),(2,E'),(3,+),(4,F),(5,E'),(6,+),(7,F),(8,E')}
E(T) = {(0,1),(0,2),(2,3),(2,4),(2,5),(5,6),(5,7),(5,8)}
T = <R(T),E(T),L(T)>

Grammatiche e Alberi di Derivazione Sintattica

Alberi di derivazione sintattica

Definizione. Immagine I(G) della chiusura transitiva, =>*, della relazione binaria su T(G), "=>", definita dalle produzioni di una grammatica.

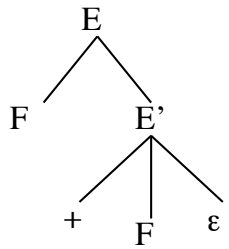
Dove: $G \equiv \langle \Sigma, V, s \in V, P \equiv \{x_i ::= x_{i,1} \ldots x_{i,ji} \mid i \leq |P|\} \rangle$

$T(G) \equiv \langle R, E \subseteq R^2, L: R \to \Sigma \cup V \rangle$ *s.t.:* $|R| \geq |\Sigma \cup V|$

$E \equiv \{(l,r_1),\ldots,(l,r_h) \mid L(l)=x \;\&\; L(r_i)=x_i \;\&\; x ::= x_1 \ldots x_h \in P\}$

$=> \equiv \{(T,U) \mid arity(root(T))=0 \& L(root(T))=x=L(root(U)) \& L(sons(U))=x_1 \ldots x_j$
per $x ::= x_1 \ldots x_j \in P$, *oppure*
$L(root(T))=L(root(U)) \& son(T,j)=son(U,j)\ (\forall j \in 1..k \setminus i) \&$
$son(T,i) => son(U,i)$ *per* $1 \leq i \leq k \equiv arity(root(T))\ \}$

I(G) $\equiv \{T \mid T_s =>^* T \;\&\; arity(root(T_s))=0 \;\&\; L(root(T_s))=s\}$



E::= F E'
E'::= + F E'
E'::= ε

AST: Alberi di Sintassi Astratta

Alberi di Sintassi Astratta

Definizione. Relazione binaria, antisimmetrica, "$>$" su Termini s.t:

$$f_k(t_1,\ldots,t_k) > t_i \qquad (\forall\, 1 \le i \le k \;\&\; f_k(t_1,\ldots,t_k), t_i \in \mathit{Termini})$$

Equivalentemente: $> \equiv \{(f_k(t_1,\ldots,t_k),t_i) \mid 1 \le i \le k,\ f_k(t_1,\ldots,t_k),t_i \in T_\Sigma\}$

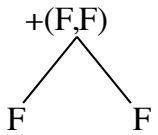
Dove: $T_\Sigma \equiv \{f_k(t_1,\ldots,t_k) \mid f_k \in \Sigma \;\&\; t_1,\ldots,t_k \in T_\Sigma\}$ *--- Termini su* Σ

Equivalentemente: $> \equiv \langle R, E \subseteq R^2, L{:}R \to T_\Sigma \rangle$ *s.t.:* $|R| \ge |T_\Sigma|$

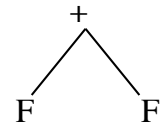
Dove: $E \equiv \{(l, r_1),\ldots,(l, r_k) \mid L(l) = Opt(f_k(t_1,\ldots,t_k)) \;\&\; L(r_i) = Opt(t_i)\ (\forall f_k(t_1,\ldots,t_k) \in T_\Sigma)\}$

$\Sigma = \{F_0, +_2\}$
$T_\Sigma ::= \{F\} \cup \{+(u,t) \mid u,t \in T_\Sigma\}$

Opt: $T_\Sigma \to \Sigma$
$Opt(f_k(t_1,\ldots,t_k)) = f_k$



oppure



con una differente
funzione etichetta L

Approccio
Semantic Attachment

ATTRIBUTE GRAMMARS e SYNTAX DIRECTED DEFINITIONS (translations)

simboli grammaticali (di ogni derivazione) + attributi

tree(E) ovvero E.tree

E.depth

Valori (di opportuni **Domini**) Identificabili (riferibili) attraverso **Nomi**

SDD = grammatica +
funzione che associa:
attributo p al simbolo A (A.p)
espressione che definisce valore di A.p

una semplice Grammatica G context free

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

Una **funzione** che associa valori agli **attributi** dei simboli

Una grammatica ad attributi per G

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:=E'.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,**

Come si usano le SDD

- Proprieta' delle SDD
- Come e quando si calcolano i valori degli attributi
  - Attribute Grammars: Semantica
  - Grafo delle dipendenze e Dgraph Topological Sort
- Due classi di attributi
- Tre tipi di sistemi di calcolo

Le proprieta' delle SDD

A::= β X1.p1:=e1,..., Xn.pn:=en

1) gli attributi di un simbolo sono **definiti localmente alla produzione** in cui occorre il simbolo

$$\{X1,...,Xn\} \subseteq \{A\} \cup S(\beta)$$

A::= B a A | b {C.c:=C.c+1}
B::= C B | d
C::= c C | c

NO

dove S(β) e' l'insieme dei simboli occorrenti in β.

A::= β X1.p1:=e1,..., Xn.pn:=en

2) le **espressioni** con cui sono definiti possono utilizzare solo attributi di simboli della produzione

$$S(e1) \cup ... \cup S(en) \subseteq \{A\} \cup S(\beta)$$

A::= B a A | b {A.c:=C.c}
B::= C B | d
C::= c C | c

NO

A_1::= B a A_2 | b {A_1.c:=B.c + A_2.c}
B::= C B | d
C::= c C | c

Se una produzione ha piu' occorrenze di uno stesso simbolo grammaticale:

indiciamo occorrenze differenti per distinguere gli attributi di una da quelli dell'altra occorrenza

A::= A B A diventa: A_1::=A_2 B A_3

pertanto A_3.d:= A_1.d+A_2.d+B.d

A::= β X1.p1:=e1,..., Xn.pn:=en

3) epressioni possono usare

operatori (calcolare valori)

attribute grammars

effetti laterali (modificare lo stato)

print

update di symbol-table

Attenzione al metalinguaggio con cui esprimiamo le definizioni degli attributi

A_1::= B a A_2 | b {if B.c > A_2.c then A_1.c:=A_2.c else A_1.c:=B.c }
B::= C B | d
C::= c C | c

A_1.c:=min(A_2.c, B.c)

A::= β X1.p1:=e1,..., Xn.pn:=en

4) espressioni devono essere effettivamente calcolabili

X.p:= *e*(Y1.p1,...,Yk.pk)

Allora:

- operatori in *e* tutti calcolabili (meta è un L.P.)
- valori di Y1.p1,...,Yk.pk devono essere tutti disponibili quando calcoliamo X.p

A_1::= B a A_2 | b {A_1.c:=A_1.c + B.c}
B::= C B | d
C::= c C | c

{B.c:=A_1.c, A_2.c:= B.c}

Grammatica ad Attributi: Semantica

$T = \langle R, E \subseteq R^2, L: R \to U \rangle$ --- *etichettati su U*

$T^A = \langle T, L_{\{ai\}}: R(T) \to U_{\{ai\}} \rangle$ --- *con attributi $\{a_i\}$ a valori su $U_{\{ai\}}$*

Grammatica: $G = \{\Sigma, V, s, P\}$

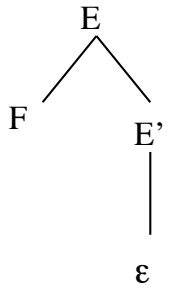
- $P \subseteq V \times E_{\Sigma+V}$
- $L(G) = I(G)$

Grammatica Attributata: $G^A \equiv \{\Sigma, V, s, P^A, \{a_i\}\}$

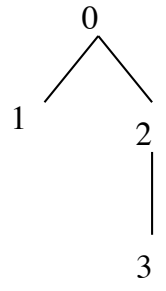
- $G^A\downarrow \equiv \{\Sigma, V, s, P\} = G$, *con $P = P^A\downarrow$*
- $P^A \equiv P \times (\{a_i\} \times Meta_{\{ai\}})$
- $L(G^A) \equiv \{\langle T, L_{\{ai\}} \rangle \mid T \in L(G^A\downarrow) \;\&\; \forall r_0 \in R(T)$:
  - $sons(r_0) = r_1, \ldots, r_n, (\forall i \in 0..n) L(r_i) = B_i, B_0 ::= B_1, \ldots, B_n \{B_i.a_{ji} = e_{i,ji}\}^{i \in 0..n} \in P^A$
  - $L_{aj}(r_i) \equiv Sem_{Meta}(e_{i,j}\,[L_{ak}(r_h)/B_h.a_k]^{(h \neq i) OR (k \neq j)})\}$
    - *$E[V/x]$* significa *E* con *x* rimpiazzata da *V*
    - *$F^{c(i)}$* significa più occorrenze di *F* ciascuna con *i* rimpiazzato da un valore per cui *c(i)*

Grammatica ad Attributi: Applichiamo la Semantica

E::= F E' {E'.d=E.d+1}
E'_1::= + F E'_2 {E'_2.d=E'_1.d+1}
E'::= ε {ε.d=E'.d}



parse tree

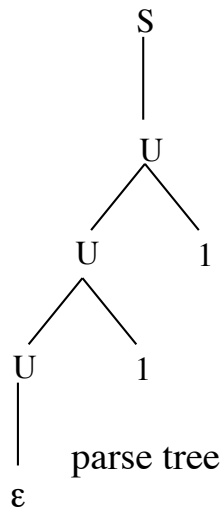


Il vero albero T

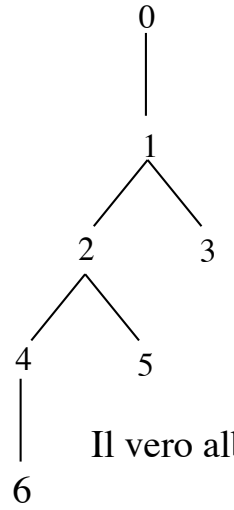
T^A={<R={0,1,2,3}, E, L={(0,E),(1,F),(2,E'),(3,ε)}>,
L_d={(0,Sem_{Meta}(⊥)),
(2,Sem_{Meta}((E.d+1)[L_d(0)/E.d])),
(3,Sem_{Meta}((E'.d)[L_d(2)/E'.d]))}}

Applichiamo la Semantica: Un altro esempio [Aho-pag.316]

S::= U {U.count:=0}
U::= U_1 1 {U_1.count=U.count+1}
U::= ε {print(U.count}



parse tree



Il vero albero T

T^A={<R={0,1,2,3,4,5,6}, E, L={(0,S),(1,U),(2,U),(3,1),(4,U),(5,1),(6,ε)}>,

L_{count}={(0,Sem_{Meta}(⊥)), (1, Sem_{Meta}(0)), (2,Sem_{Meta}((U.count+1)[L_{count}(1)/U.count])),
(3,Sem_{Meta}(⊥)), (4,Sem_{Meta}((U.count+1)[L_{count}(2)/U.count])), (5,Sem_{Meta}(⊥)),
(6, Sem_{Meta}(print(U.count)[L_{count}(4)/U.count]))}

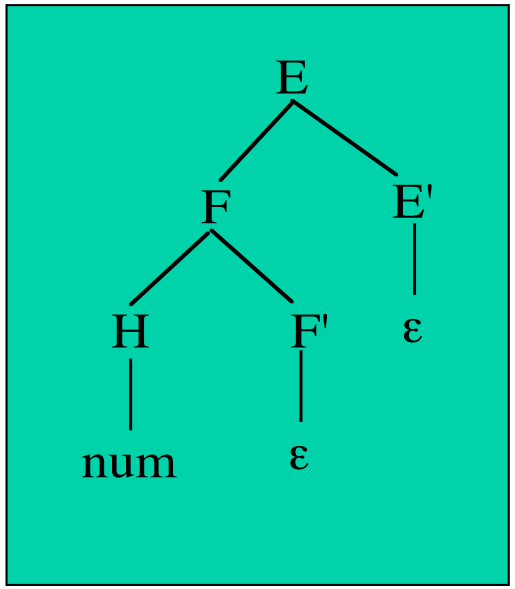
Sem_{Meta}(print(U.count)[L_{count}(4)/U.count]) = Sem_{Meta}(print(U.count+1)[L_{count}(2)/U.count])
= Sem_{Meta}(print(U.count+1+1)[L_{count}(1)/U.count])
= Sem_{Meta}(print(0+1+1))

GRAFO DELLE DIPENDENZE

X.p:= e(Y1.p1,...,Yk.pk)

Y1.p1,...,Yk.pk occorrenti nel parse tree devono essere legate prima di X.p

G.D. = 1 vertice per ogni attributo
1 arco per ogni coppia di attributi u.p e v.q
orientato da v.q a u.p se u.p *dipende* da v.q



Parse Tree

E::=F E' {E.tree:= mk-tree('E', F.tree, E'.tree), F.depth:=E.depth+1, E'.depth:=E.depth+1}

...

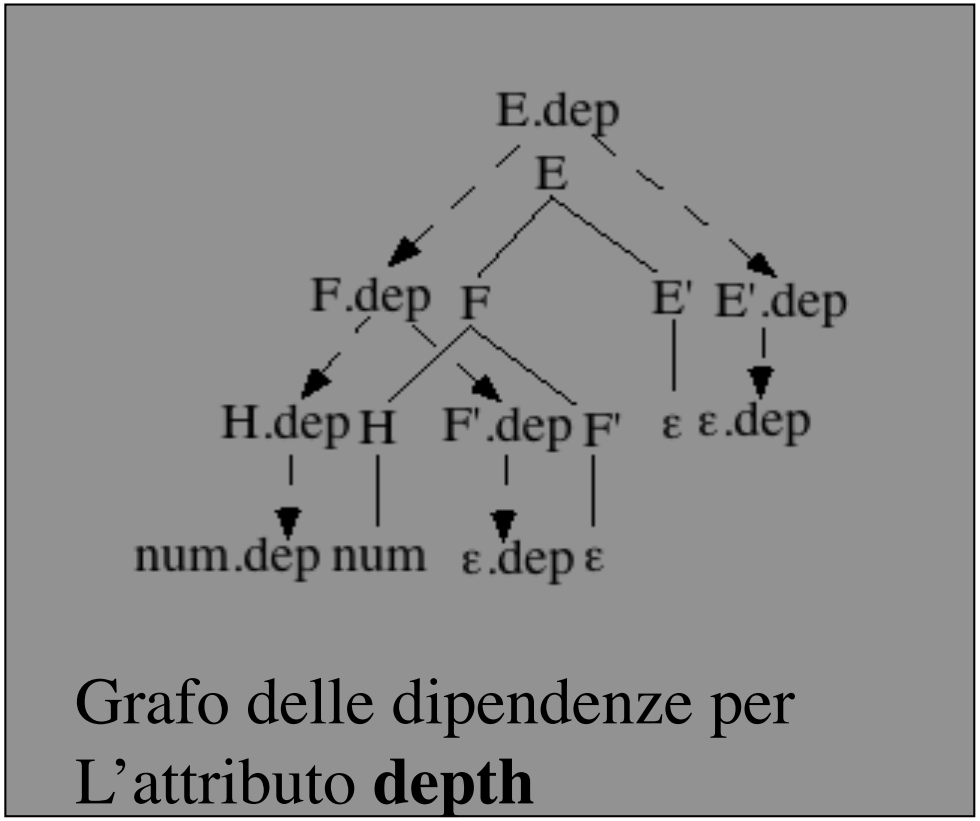
E'::=ε {E'.tree:= mk-tree('E'', mk-leaf('ε')), ε.depth:=E'.depth+1}

F::=H F' {F.tree:= mk-tree('F', H.tree, F'.tree), H.depth:=F.depth+1, F''.depth:=F.depth+1}

...

F'::=ε {F'.tree:= mk-tree('F'', mk-leaf('ε')), ε.depth:=F'.depth+1}

H::=num {H.tree:= mk-tree('H', mk-leaf('num')), num.depth:=H.depth+1}



TOPOLOGICAL SORTING di un grafo =

ordinamento *parziale* dei nodi che rispetta la relazione definita dall'orientamento degli archi: *no cicli*

quale dei due grafi contiene espressioni calcolabili in modo effettivo ?

la prima perche'.....

mentre la seconda....

Useremo sempre G.D. + T.S. per stabilire effettivita'

T.S. fornisce anche un *controllo per l'esecuzione* delle espressioni

controllo vincolato al calcolo del parse tree

Per calcolare gli attributi
dobbiamo sempre generare prima il parse tree ?

NO: vedremo quando possiamo evitarlo

- Vantaggioso in generale (anticipiamo il calcolo)
- Indispensabile in compilatori 1-passo

VARI TIPI DI ATTRIBUTI

sintetizzato
dipende da attributi dei soli figli nel parse tree

$X ::= \beta \qquad X.p := e(Y1.p1,\ldots,Yk.pk)$

dove $\{Y1,\ldots,Yk\} \subseteq S(\beta)$

ESEMPIO: tree

$A_1 ::= B\ a\ A_2$ {$A_1.c := B.c + A_2.c$}
A::= b {A.c:= …}
$B_1 ::= C\ B_2$ {$B_1.c$:= …}
B::= d {B.c:= …}
C::= c C | c

Utili in: analisi e/o semantica composizionale
semantica costrutto dipende solo da
semantica delle strutture componenti

ereditato

dipende da attributi del padre e dei fratelli nel parse tree

$A ::= \beta$ $\quad X.p := e(Y1.p1,...,Yk.pk)$

dove $\{Y1,..,Yk\} \subseteq \{A\} \cup S(\beta)$ & $X \in S(\beta)$

ESEMPIO: depth

A::= B a A {B.inc:=0}

A::= b

B_1::= C B_2 {B_2.inc:=C.outc, B_1.outc:=B_2.outc}

B::= d {B.outc:=B.inc}

C_1::= c C_2 {C_1.outc:=...}

C::= c {C.outc:=...}

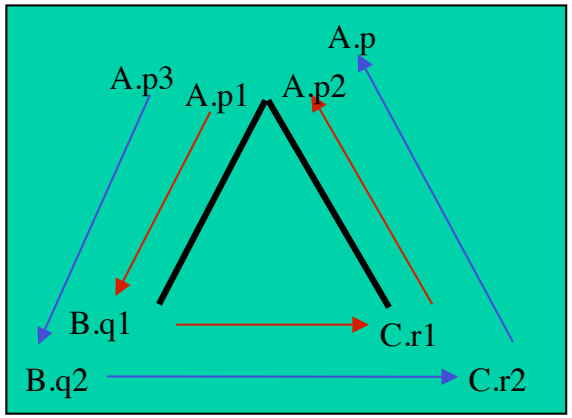
Utili per: esprimere proprietà che dipendono dal contesto in cui il costrutto occorre e/o semantica non-composizionale

esempio: distinzione tra espressioni denotabili e memorizzabili

VALUTAZIONE DEGLI ATTRIBUTI

Tre metodi per calcolare le *regole semantiche*

1) **parse tree**:
   - si costruisce il parse tree P
   - si costruisce G.D.
   - si genera un T.S.
   - si costruisce una procedura che visita P in accordo a T.S. e valuta le espressioni (azioni) associate



inconvenienti: almeno due passi (per le quattro fasi)
occorre costruire il parse tree interamente

2) **rule-based**:

- il controllo nell'ordine della valutazione è stabilto ispezionando la SDD e inserito in una co-routine
- la co-routine è eseguita alternandosi al parser

inconvenienti: scarsa semplicità e error prone

(spazio nomi Parser e SDD, scelta dei tempi di interazione)

3) **oblivious**:

il controllo è stabilito dal parser ed è indipendente dagli attributi

inconvenienti: scarsa applicabilità

(il contollo del parser deve essere compatibile con un T.S.)

2-3 sono a compile construction time
1 è in tempo successivo

Applicazioni delle Attribute Grammars

- Potenza: context sensitive e attribute grammars
- Il metodo oblivious: visita depth-first
- Grammatiche l-attributate
- Esecutori bottom-up: sintetizzati
- Esecutori top-down: l-attributate
- Bottom-up: Trasformazioni per l-attributate

Le attribute grammars (o SDD) hanno una notevole potenza

ricordate !! : un linguaggio non context-free

$L_2=\{u^n v^n z^n \mid n \in N\}$

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

Il riconoscimento di grammatiche context sensitive può richiedere riconoscitori assai più complessi dei lineari LL e LR introdotti.

Utilizzando attribute grammars generiamo un riconoscitore basato su LL (LR) per L_2

Un riconoscitore basato su LL per L2

- selezioniamo un super linguaggio L1 per L2 (L2 $\subset$ L1): L1 deve avere un riconoscitore LL o LR

$$L1=\{u^n v^k z^n \mid n,k \in N\}$$

- estendiamo la grammatica G per L1 con attributi che:
  - associno al simbolo distinto S di G il valore
    vero se la frase derivata appartiene ad L2
    falso altrimenti
  - la grammatica attributata ammetta metodo oblivious

S'::= S [0]	S'.L2:= (S.n = S.k)
S^1::= u S^2 z [1]	S^1.n:=S^2.n +1 , S^1.k:=S^2.k
S::= L [1]	S.n:= 0 , S.k:=L.k
L^1::= v L^2 [2]	L^1.k:=L^2.k +1
L::= ε [3]	L.k:=0

Quale linguaggio genera la grammatica: ovvero chi è l'insieme frontiera dei parse tree irradicati su S', PT(S'),

$$L1=\{ \mathit{fron}(T) \mid T \in PT(S')\}$$

$$L2 = \{ \mathit{fron}(T) \mid T \in PT(S') \ \&\ S'.L2\}$$

Abbiamo un riconoscitore per L2 senza utilizzare
né una grammatica *context sensitive*
né un riconoscitore di tale tipo

L'attributo L2:

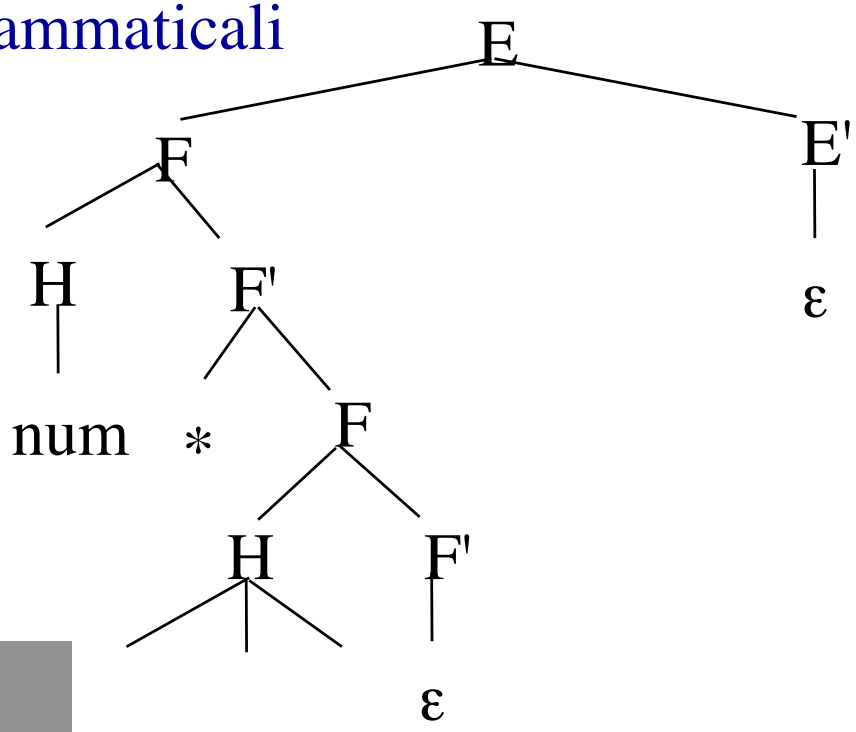
di che tipo è ?

è valutabile utilizzando oblivious
basato su parsing di G ?

Metodo oblivious: quando e' applicabile ?

compatibilita' tra
ordine di valutazione attributi e
generazione simboli grammaticali

Top-down e Bottom-up
generano sempre depth-first:



Il **metodo oblivious** è sempre utilizzabile quando *depth-first* è un *topological sort* per il *grafo delle dipendenze* della grammatica

GRAMMATICHE L-ATTRIBUTATE

SDD è l-attributata se in ogni produzione

$$A0 ::= A1 \ldots Ak$$

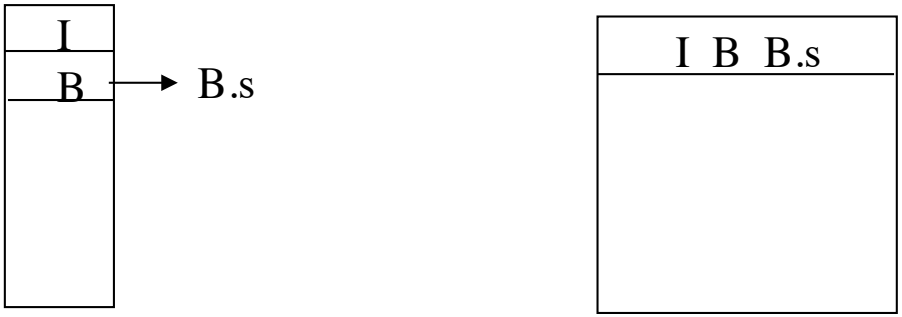
gli ereditati di Aj, per $1 \leq j \leq k$, dipendono da attributi di Ai con $0 \leq i < j$

le **S-attributate** (grammatiche con soli sintetizzati) sono l-attributate

Th. Se G ha riconoscitore top-down/bottom-up e G' è un l-attributata di G. Allora, G' ha esecutore oblivious top-down/bottom-up.

Esecutori Bottom-Up per S-attribute

- estendiamo i valori dello stack di controllo di LR
  associamo ad ogni simbolo:
  - gli eventuali attributi (sintetizzati)
  - lo stato della transizione dell'automa dei viable prefix



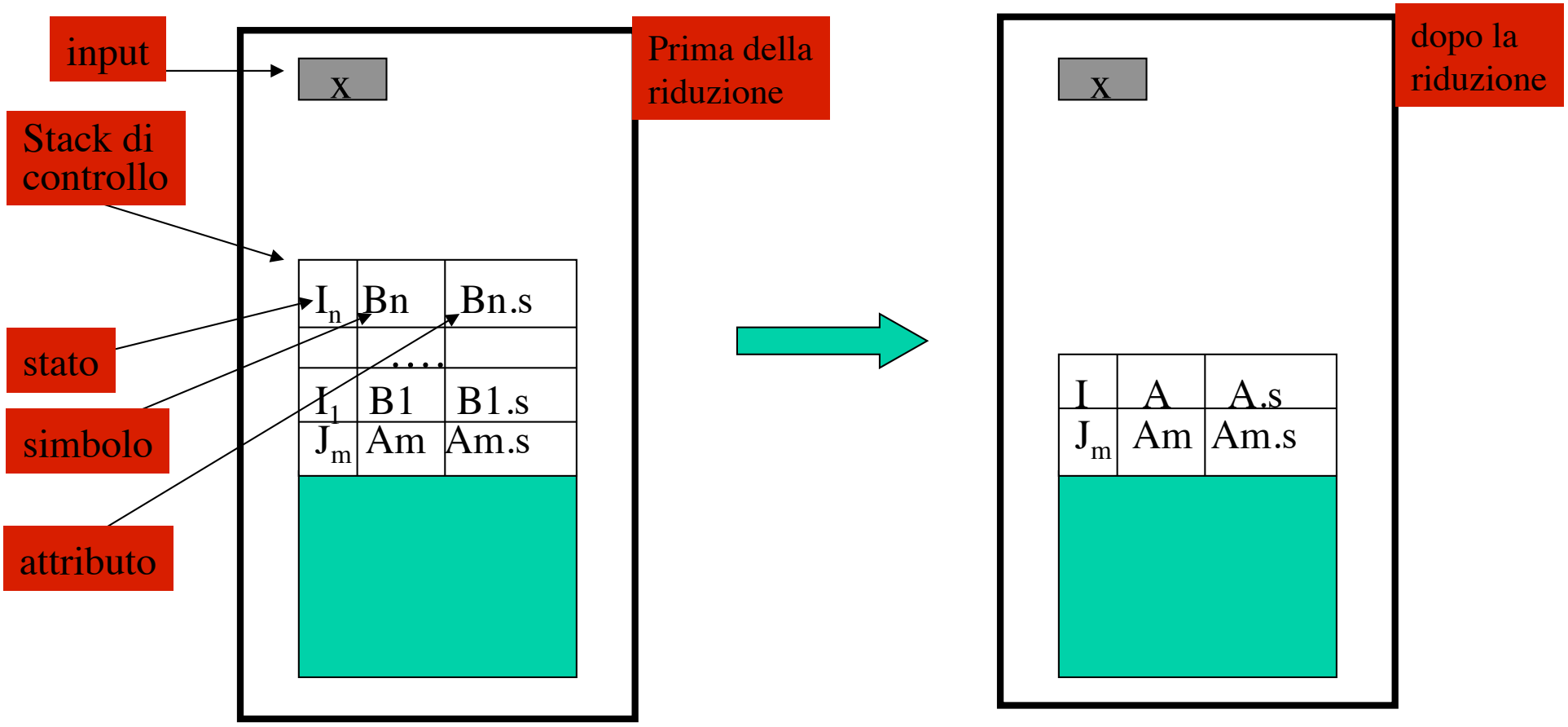
- Ad ogni riduzione A::=B1…Bn {α} calcoliamo l'azione definente i sintetizzati di A.
  Se α richiede i valori di sintetizzati di Bi,
  tali valori sono associati a Bi che si trova
  (n-i) posizioni dal top dello stack

produzione

(k) A::=B1…Bn {α}

Tabelle riconoscitore LR

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



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

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

Esecutori Top-Down per L-attribute

- Facciamo emergere le dipendenze tra attributi

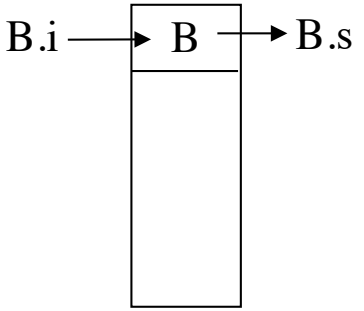
translation scheme = grammatiche c.f. le cui produzioni contengono semantic actions (espressioni) che calcolano attributi dei simboli grammaticali

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

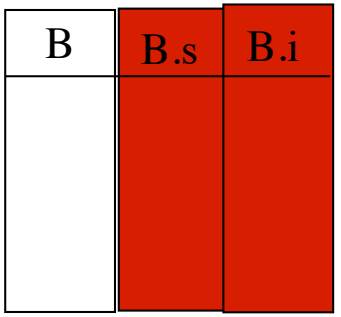
- ereditato di un simbolo destro, Bi, deve essere calcolato in action, βi, che precede il simbolo
- action puo' usare solo attributi di simboli Bi che precedono e gli ereditati di A

Th. Ogni L-attributata puo' essere trasformata in un T.S.

- Trasformiamo la L-attributata in un T.S.



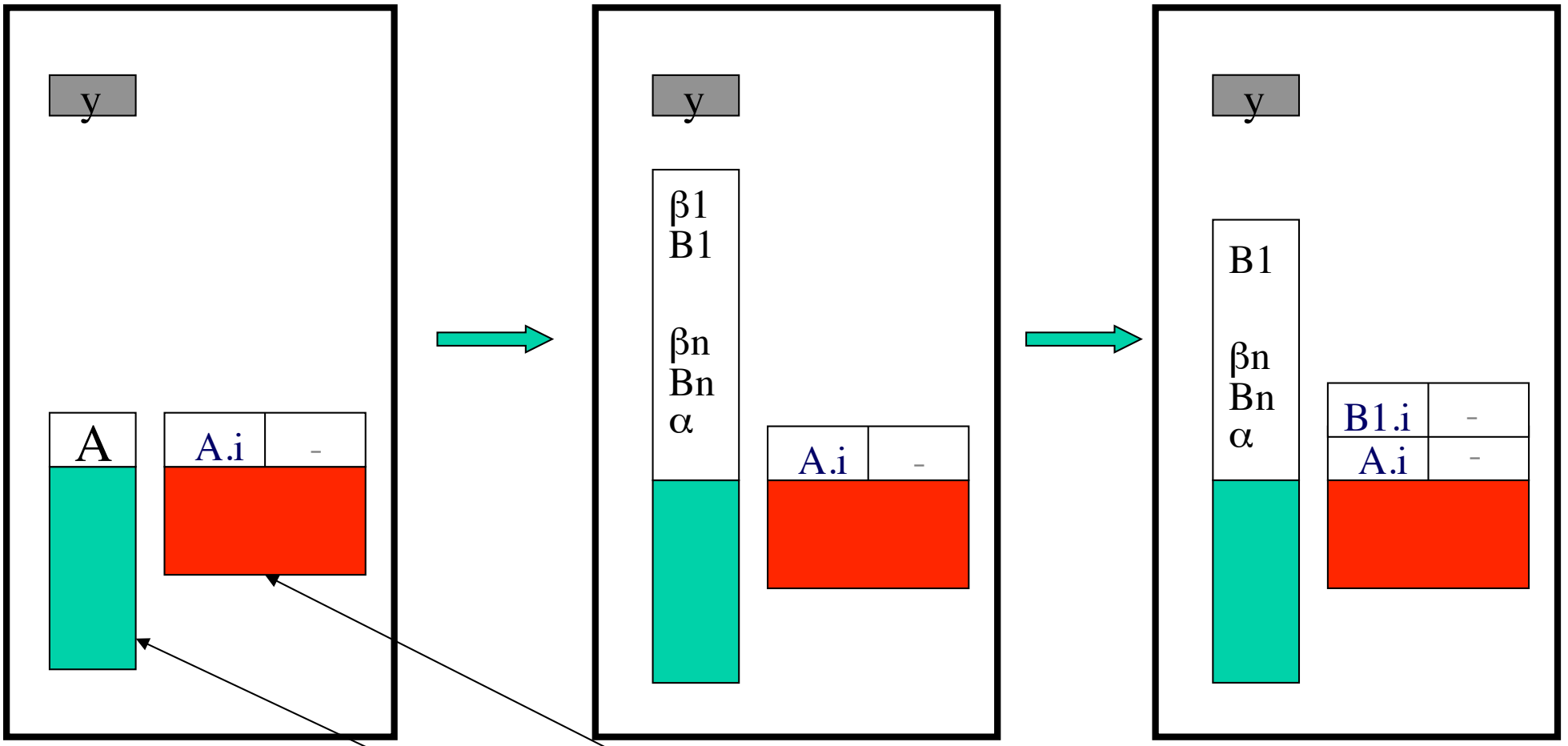
- Accoppiamo lo stack di controllo, C, con
  - uno stack per i sintetizzati, S,
  - uno stack per gli ereditati, I.



- Estendiamo lo stack di controllo LL con le azioni:
  - Ad ogni derivazione A::={β1}B1...{βk}Bk{α}, {β1}, B1, …,{βk}, Bk, {α} sono inserite nello stack
  - Quando un'azione, βi (o α), e' selezionata dal top dello stack, l'azione e' valutata:
    - Il valore calcolato e' posto nel top dello stack I (o S)
    - Se l'azione richiede i valori di attributi di Bj (j < i), tali valori sono estratti da (i-j) posizioni dal top dello stack

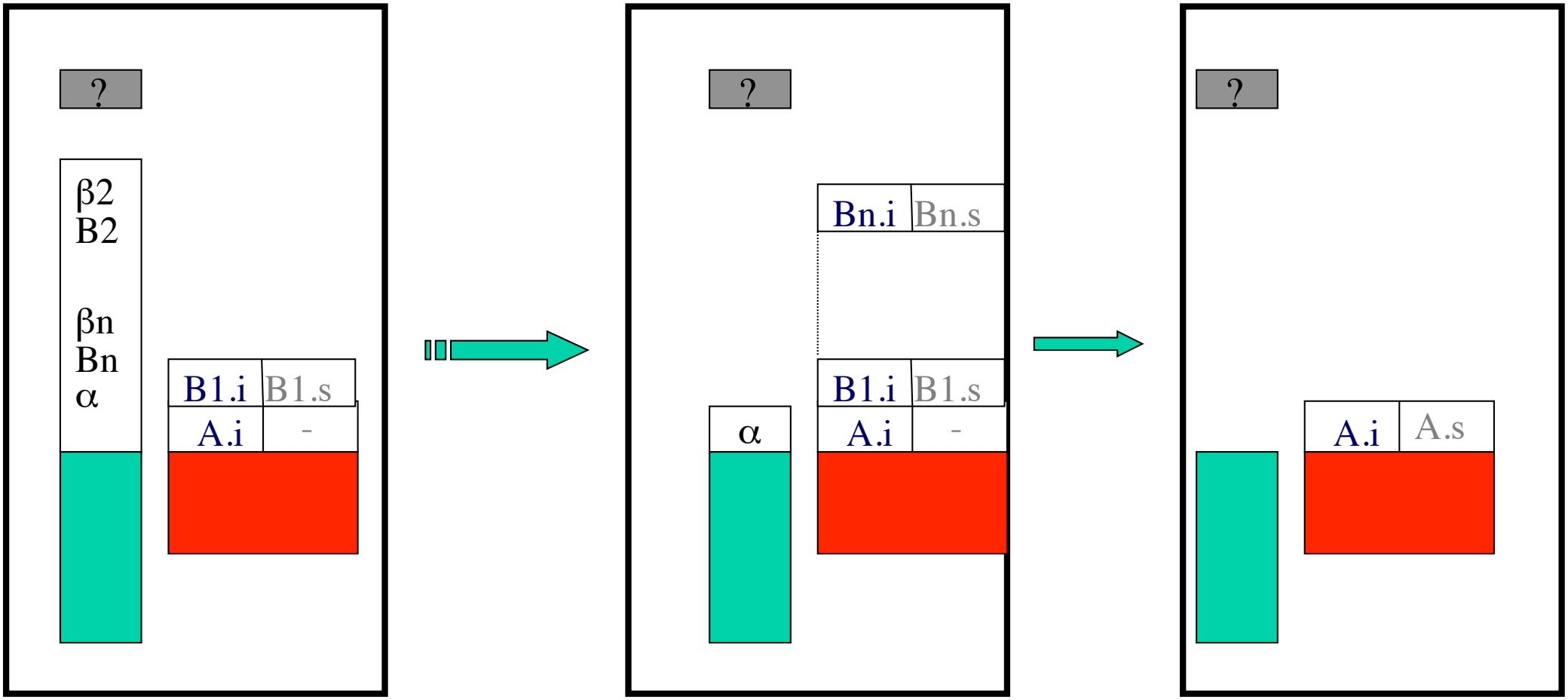
(k) A::={β1}B1…{βn} Bn {α}

M(A,y)=k



A.i e' calcolato dalla precedente derivazione (fratelli -depth first)

B1.i=[β1] e' calcolato: Puo' dipendere solo da Ereditati di A (tutti in stack)



Tutti gli attributi per calcolare α

Dove prima era A (ovvero) ora ci sono gli attributi

Bottom-up: Trasformazioni per L-attributate

MARCATORI

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

in

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

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

...

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

Schema discendente

Schema ascendente

Azioni interne allo schema sono trasformate in azioni finali su ε-riduzioni definite dai marcatori.
I marcatori, uno per azione, identificano l'origine della riduzione e consentono di identificare:
ereditati dei simboli grammaticali con *sintetizzati di marcatori*

Bottom-up: Trasformazioni per L-attributate

Fattorizzazione

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

in

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

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

...

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

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

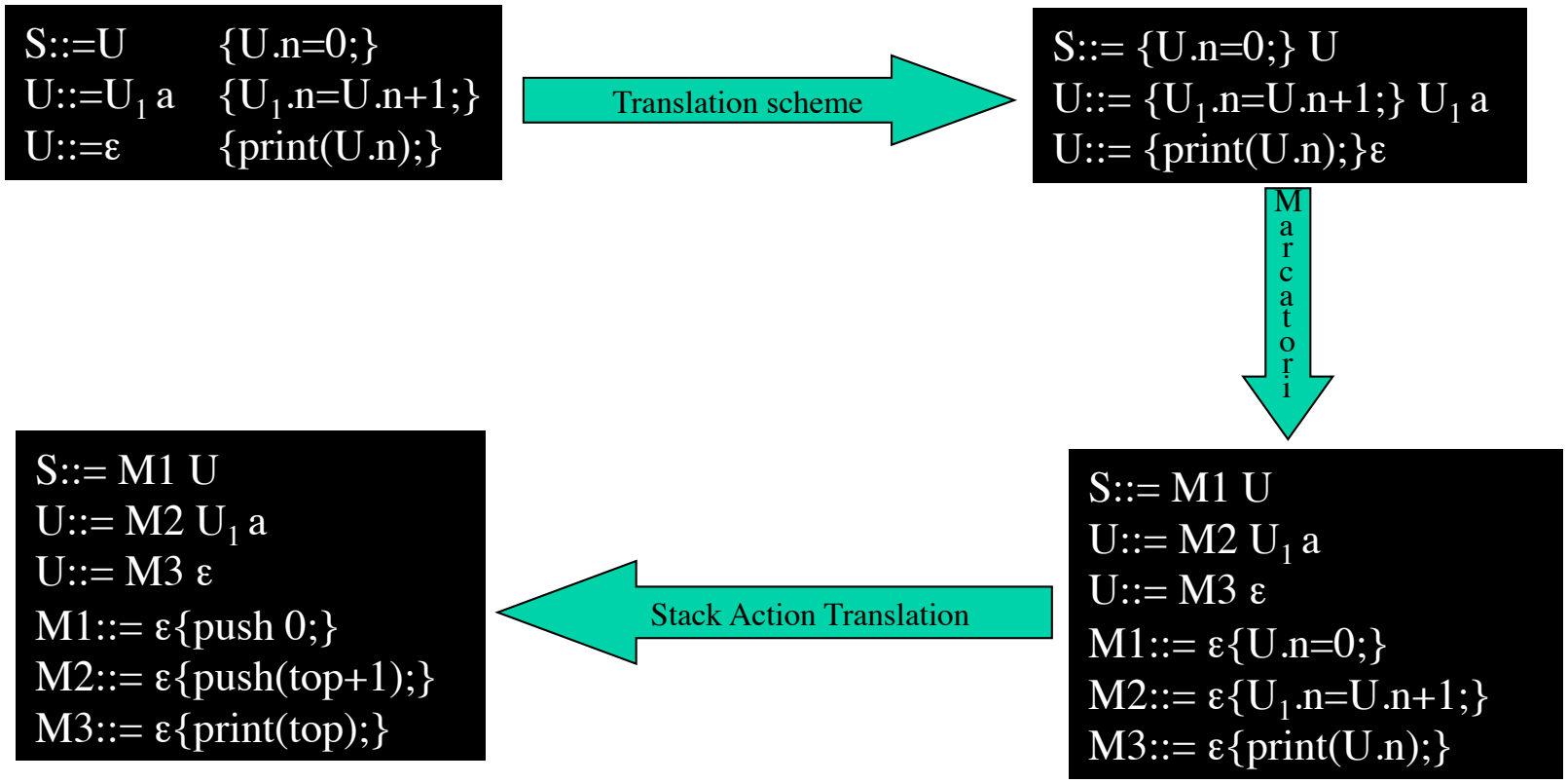
Schema discendente

Schema ascendente

Azioni interne allo schema sono trasformate in azioni finali sulle produzioni A_i. I nuovi simboli A_i associano a sinistra marcatori, uno per azione, identificano l'origine della riduzione e consentono di identificare:

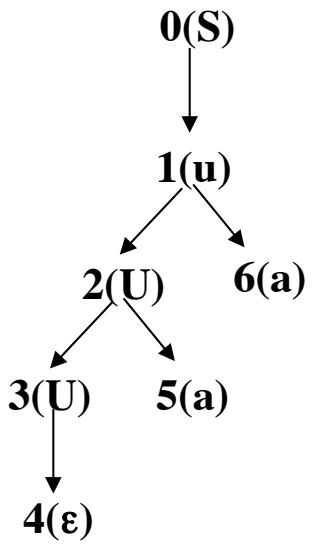
ereditati dei simboli grammaticali con *sintetizzati di marcatori*

Bottom-up: Trasformazioni per L-attribute Applichiamo (1) - Trasformazione



Bottom-up: Trasformazioni per L-attribute Applichiamo (2) - Struttura Parse Tree

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

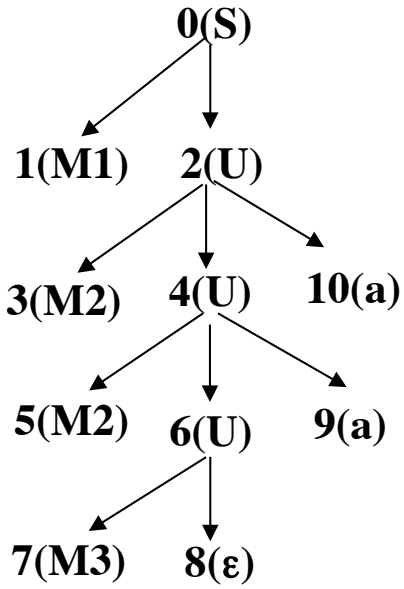


4->3->5->2->6->1->0

Analisi della frase: aa

Come cambia la visita depth-first degli alberi (postorder- posticipata)

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



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

Bottom-up: Trasformazioni per L-attributate Applichiamo (3) - Calcolo Attributi

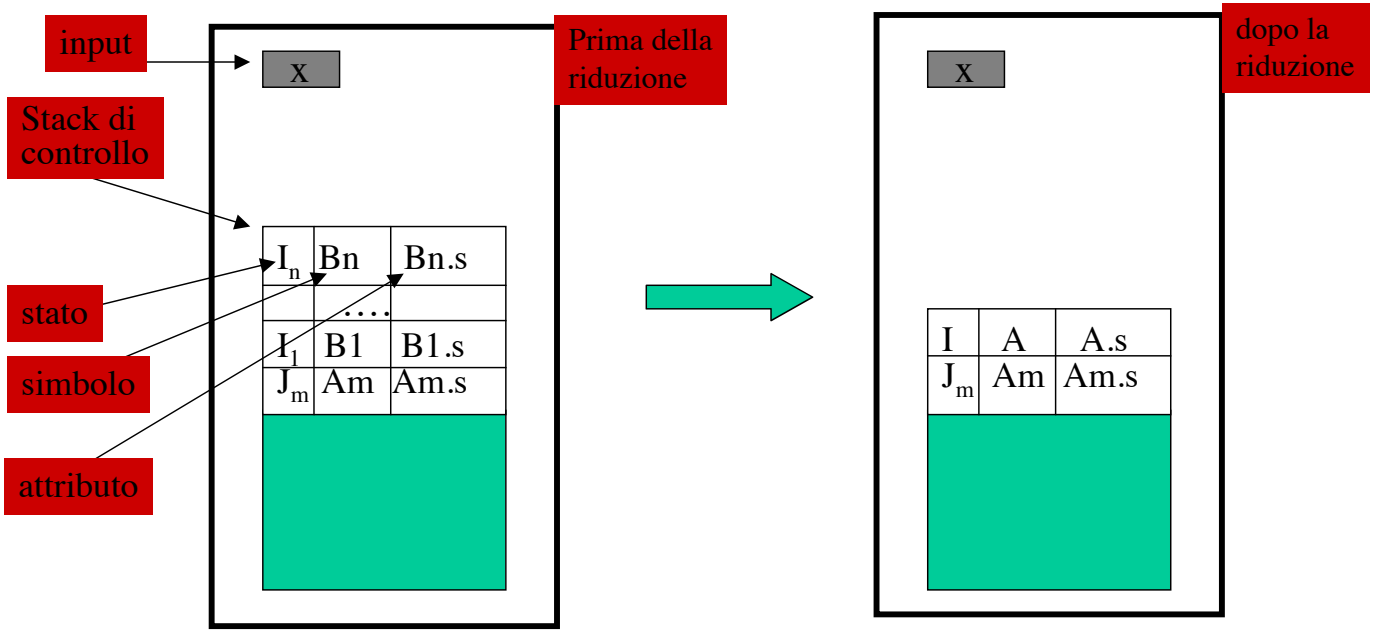
S::= M1 U
U::= M2 U_1 a
U::= M3 ε
M1::= ε{push 0;}
M2::= ε{push(top+1);}
M3::= ε{print(top);}

produzione

(k) A::=B1…Bn {α}

Tabelle riconoscitore LR

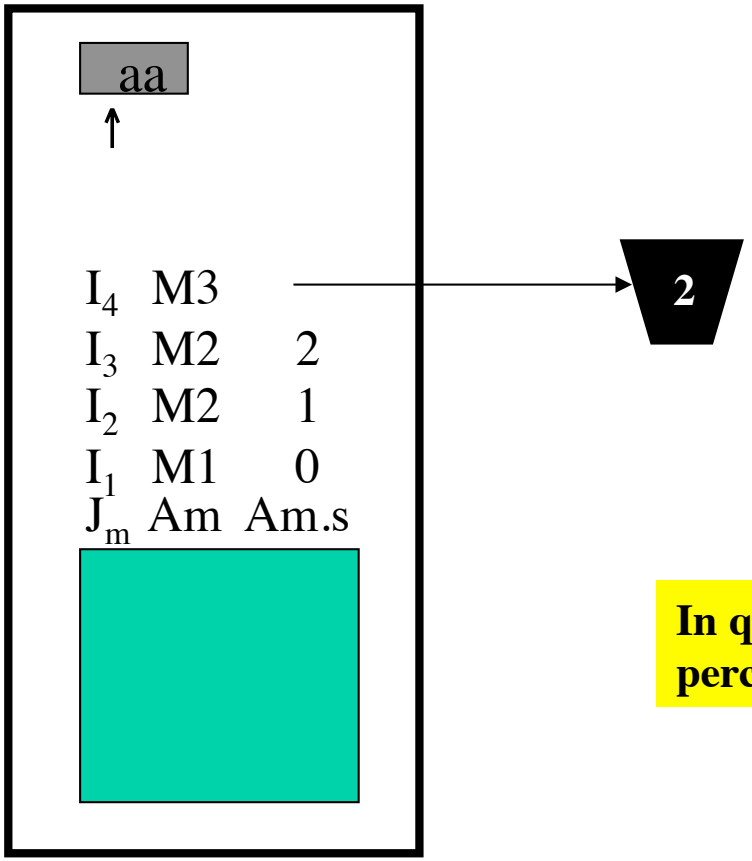
Action(I_n,x)=R/k Goto(J_m,A)=I



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

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

Bottom-up: Trasformazioni per L-attribuate Applichiamo (3) - Calcolo Attributi



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 questo caso le cose non vanno così perchè la nuova grammatica non è LR(1)

Esecutori Oblivious: Implementazione

- **Top-down**:
  - Invariante di traduzione
  - Traduzione di espressioni, α, con operandi attributi, in azioni [α] sugli stack I/S con operandi su I/S
- **Bottom-up**:
  - Invariante di traduzione
  - Traduzione di espressioni, α, con operandi attributi, in azioni [α] sullo stack di controllo C con operandi su C

argomento non trattato