Grammatiche ad attributi

Una metodologia,

una tecnica,

uno strumento per

analisi (anche contestuali) e

generazione di codice

PARSING

Riconoscitori per

appartenenza: legalita’ della frase (o stringa)

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

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

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		

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' ->

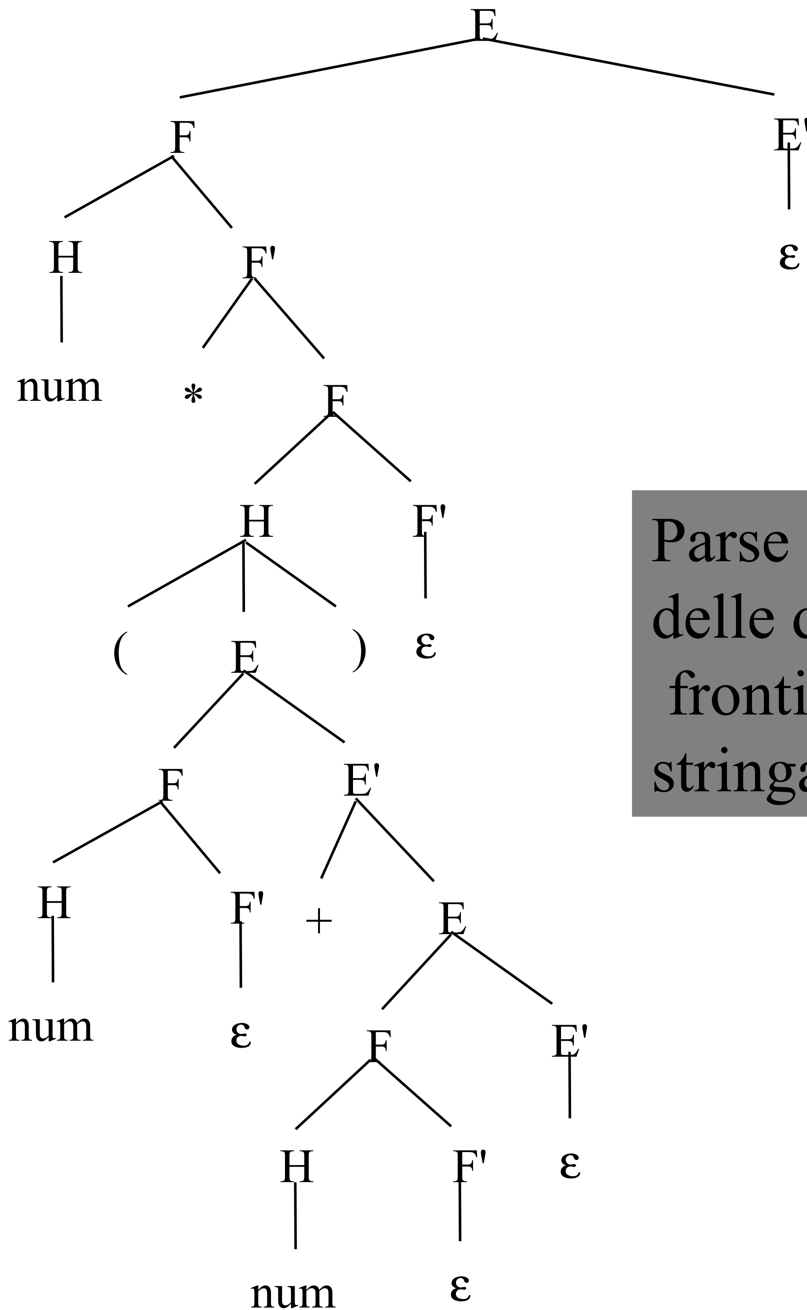
-> num*(numF'E')F'E' -> num*(numE')F'E' -> num*(num+E)F'E'

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

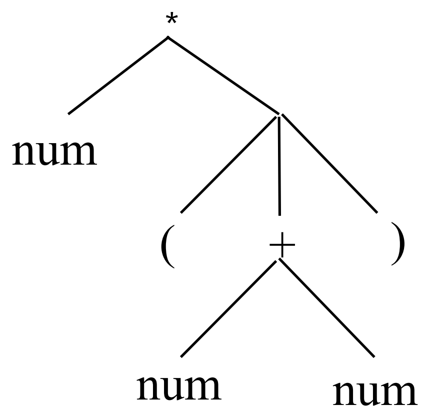
-> num*(num+numF'E')F'E' -> num*(num+numE')F'E'

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

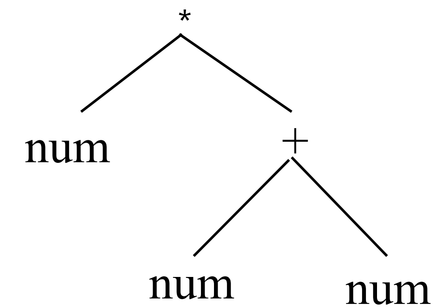
-> num*(num+num)



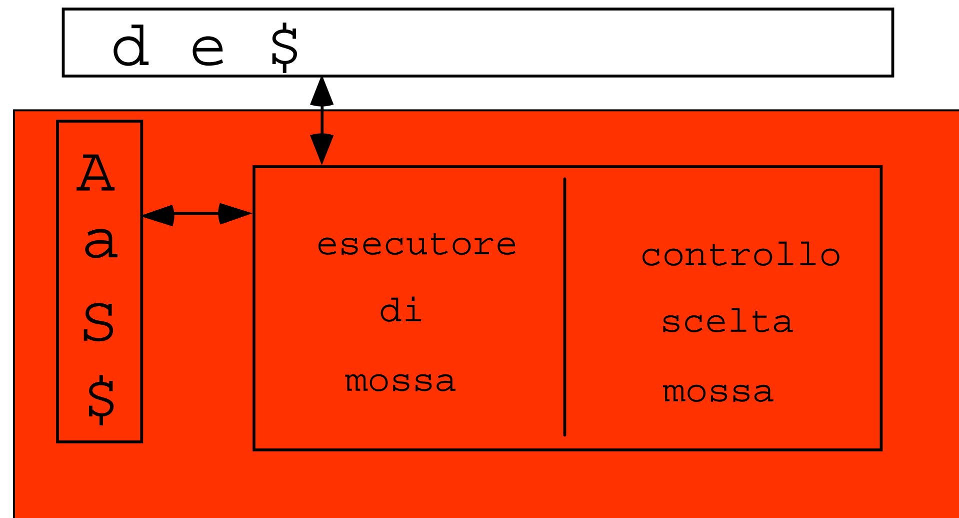
Parse Tree: mostra la struttura delle derivazioni con cui la frontiera e’ riconosciuta come stringa appartenente alla grammatica



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



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



cambiamo le mosse:

espandere

corrisponde a

creare una struttura di albero

E -> F E’

corrisponde

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

Approccio
Semantic Attachment

ATTRIBUTE GRAMMARS e SYNTAX DIRECTED DEFINITIONS (translations)

simboli grammaticali + attributi
tree(E) ovvero E.tree
E.depth

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

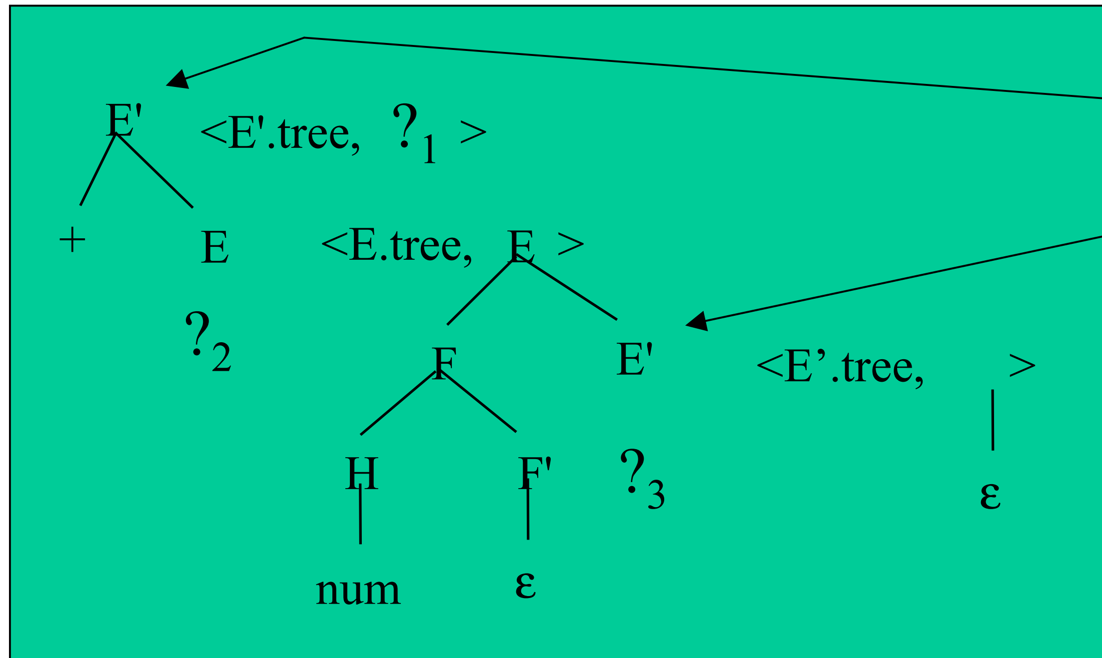
una semplice Grammatica G context free

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,**



Gli attributi sono associati a istanze dei simboli grammatical

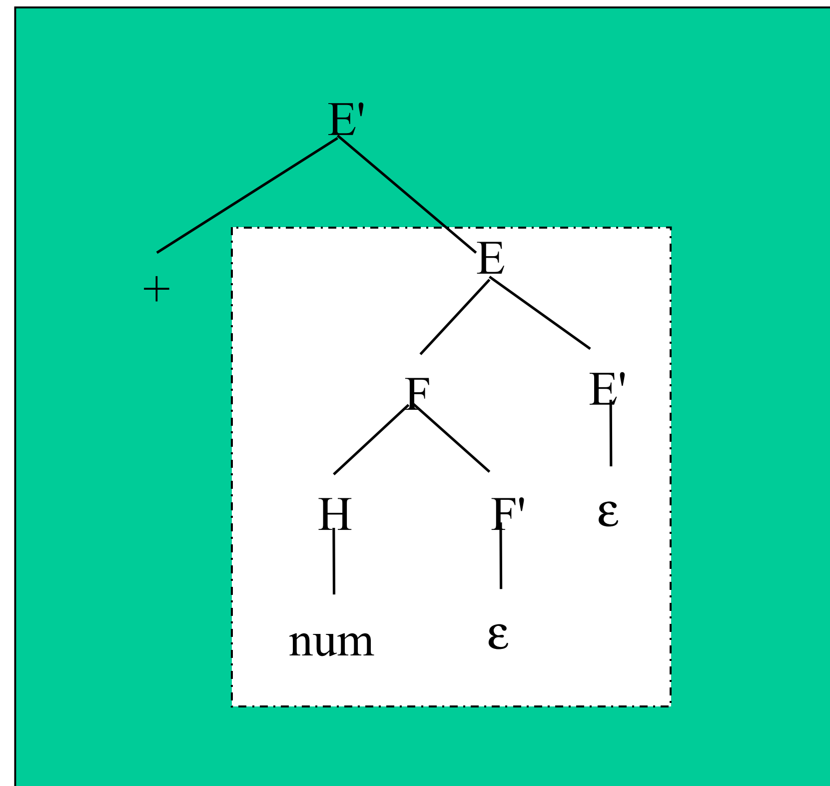
Chi e' il valore $?_1$ dell' attributo tree di E.tree ?

Chi e' l'abeto $?_2$

Chi e' il parse tree prodotto dalla derivazione:

E'-->* +num

Il parse tree della derivazione da E
coincide con
Il valore dell'attributo tree di E



Come si usano le SDD

- Proprieta' delle SDD
- Come e quando si calcolano i valori degli attributi
- Grafo delle dipendenze e 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

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

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.

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

A₁::= B a A₂ | b {A₁.c:=B.c + A₂.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

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

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

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

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

4) espressioni devono essere effettivamente calcolabili

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

allora al momento di calcolare X.p

i valori di Y1.p1,...,Yk.pk devono essere tutti disponibili

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

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

d: **QUANDO SI CALCOLA UN ATTRIBUTO?**

r: quando serve (ed a tale tempo deve essere calcolabile)

d: **Quando deve essere possibile calcolare un attributo?**

r: a chi abbiamo detto che appartiene un attributo?

ai simboli grammaticali di ogni parse tree

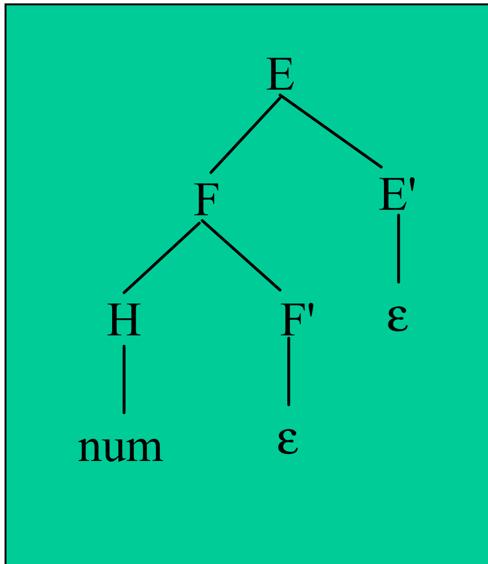
quando abbiamo i simboli grammaticali del parse tree
tutte le espressioni devono essere effettivamente calcolabili

GRAFO DELLE DIPENDENZE

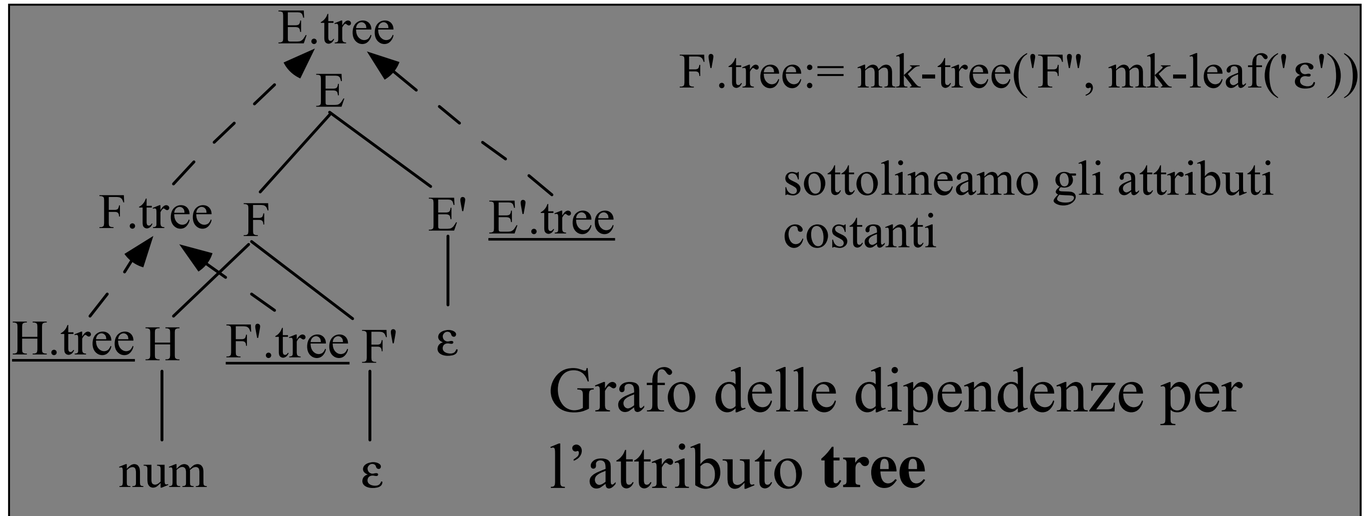
X.p:= e(Y1.p1,...,Yk.pk)

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

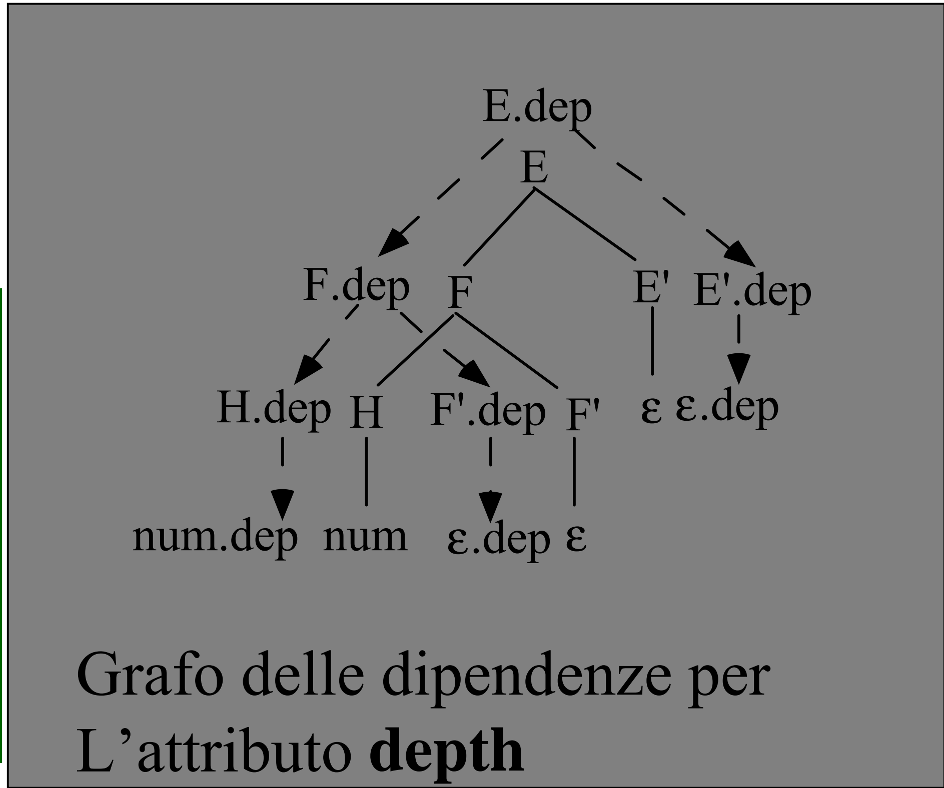
G.D. = 1 vertice per ogni attributo
1 arco per ogni coppia di attributi u.p e v.q
orientato da v.q a u.p se u.p *dipende* da v.q



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

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

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

ESEMPIO: tree

$A_1 ::= B\ a\ A_2\ \{A_1.c := B.c + A_2.c\}$

$A ::= b\ \{A.c := \ldots\}$

$B_1 ::= C\ B_2\ \{B_1.c := \ldots\}$

$B ::= d\ \{B.c := \ldots\}$

$C ::= c\ C \mid c$

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

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

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:=...}

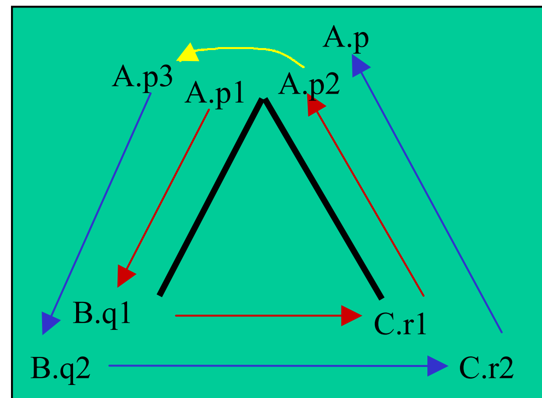
Utili per: esprimere proprieta' che dipendono dal constesto in cui il costrutto occorre e/o semantica non-composizionale

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 costrusce 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 e’stabilto ispezionando la SDD e inserito in una co-routine
- la co-routine e’ eseguita alternandosi al parser

inconvenienti: scarsa semplicita’e error prone

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

3) **oblivious**:

il controllo e’ stabilito dal parser ed e’ indipendente dagli attributi

inconvenienti: scarsa applicabilita’

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

2-3 sono a compile construction time
1 e’ 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 puo' richiedere Riconoscitori assai piu' 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]	$L_1.k := L_2.k + 1$
$L ::= \varepsilon$ [3]	$L.k := 0$

Quale linguaggio genera la grammatica: ovvero chi e' 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
ne' una grammatica *context sensitive*
ne' un riconoscitore di tale tipo

L'attributo L2:

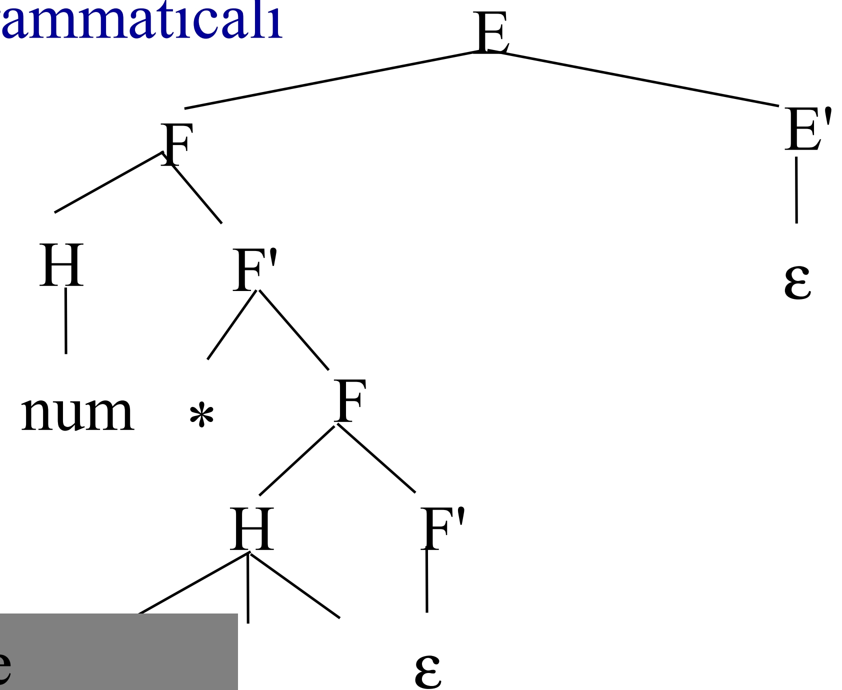
di che tipo e' ?

e' 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** e' sempre utilizzabile
quando *depth-first* e' un *topological sort*
per il *grafo delle dipendenze* della grammatica

GRAMMATICHE L-ATTRIBUTATE

SDD e' l-attributata se in ogni produzione

$$A0::=A1...Ak$$

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' e' un l-attributata di G. Allora, G' ha esecutore top-down/bottom-up.

Esecutori Bottom-Up per S-attributate

- 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

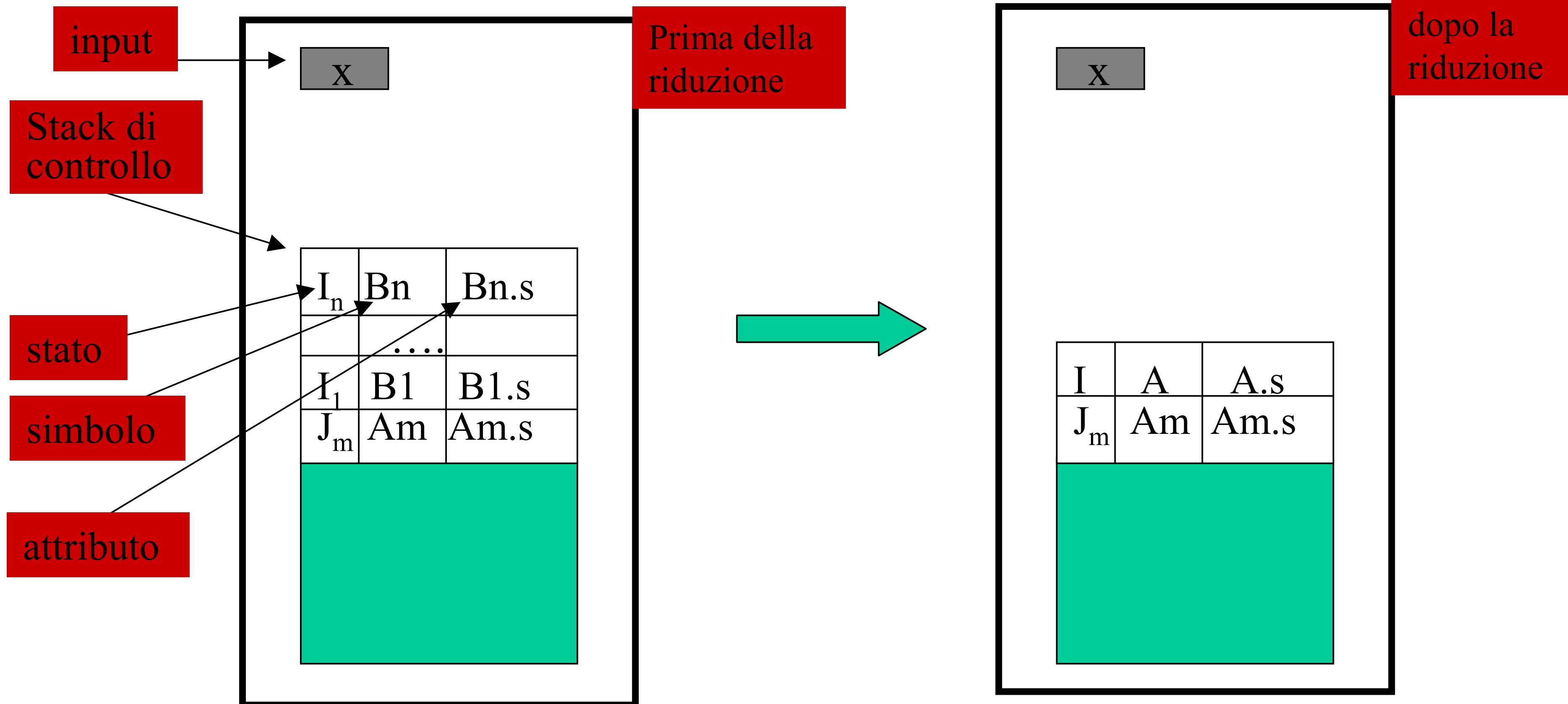
- Ad ogni riduzione A::=B1…Bn {α} calcoliamo l'azione definente i sintetizzati di A.
  Se α richiede i valori di sintetizzati di Bi,
  tali valori sono associati a Bi che si trova
  (n-i) posizioni dal top dello stack

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 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

Esecutori Top-Down per L-attributate

- Facciamo emergere le dipendenze tra attributi

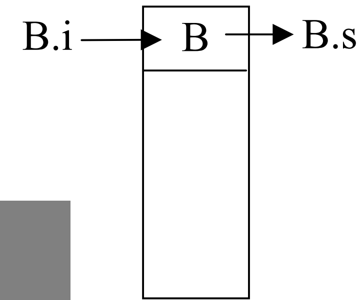
translation scheme = grammatiche c.f. le cui produzioni contengono semantic actions (espressioni) che calcolano attributi dei simboli grammaticali

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

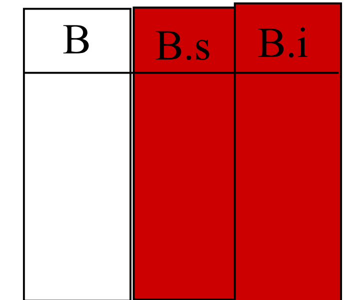
- ereditato di un simbolo destro, Bi, deve essere calcolato in action, βi, che precede il simbolo
- action puo’ usare solo attributi di simboli Bi che precedono e gli ereditati di A

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

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



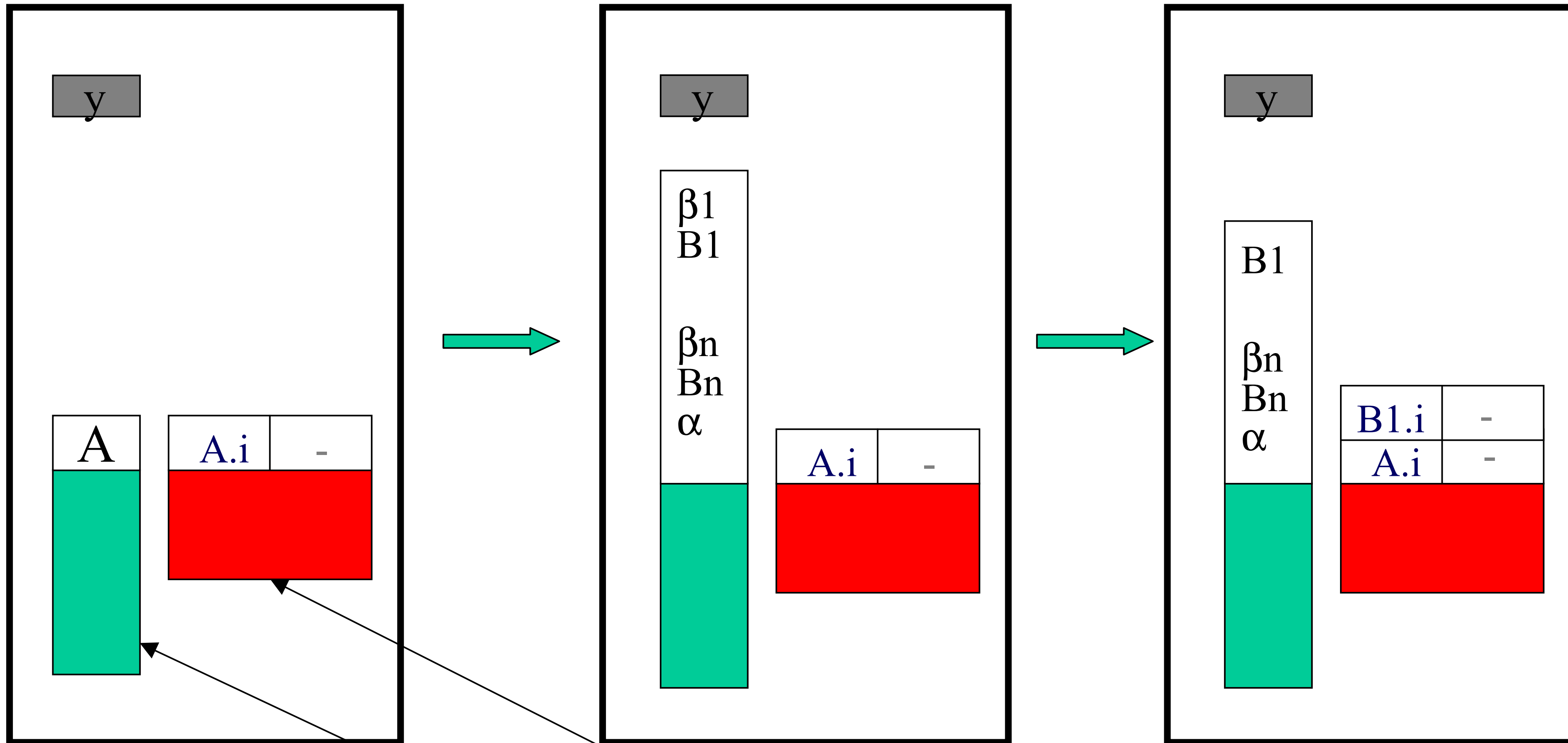
- Accoppiamo lo stack di controllo, C, con
  - uno stack per i sintetizzati, S,
  - uno stack per gli ereditati, I.



- 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

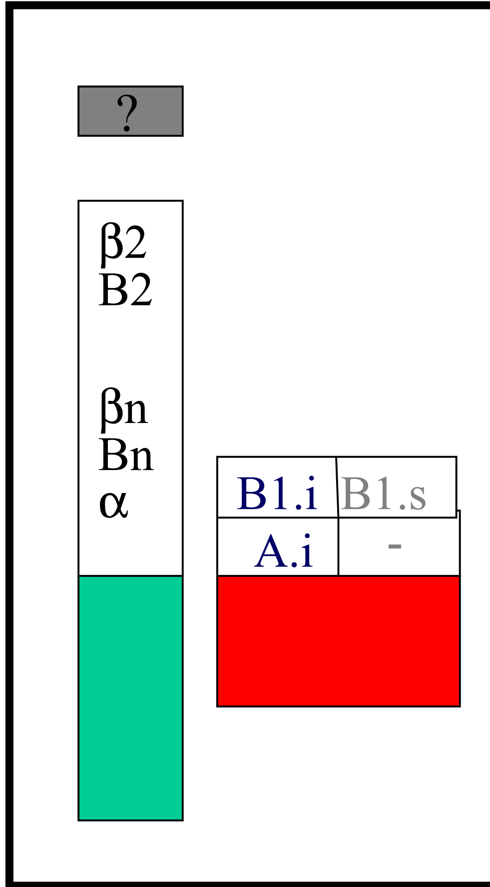
(k) A::={β1}B1…{βn} Bn {α}

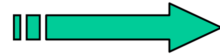
M(A,y)=k

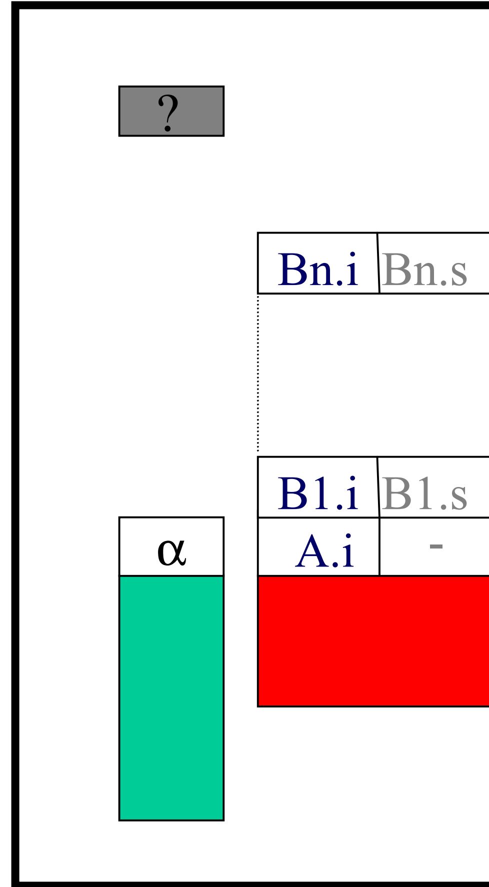


A.i e’ calcolato dalla precedente derivazione (fratelli -depth first)

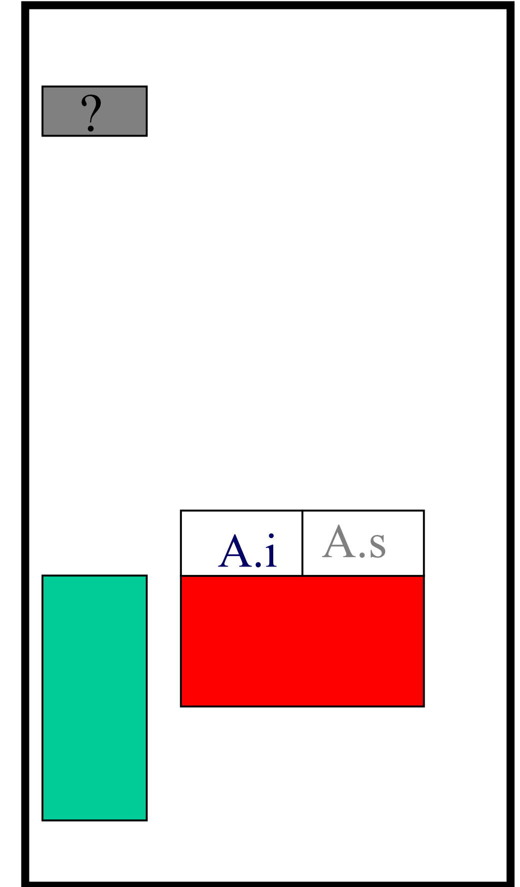
B1.i=[β1] e’ calcolato: Puo’ dipendere solo da Ereditati di A (tutti in stack)







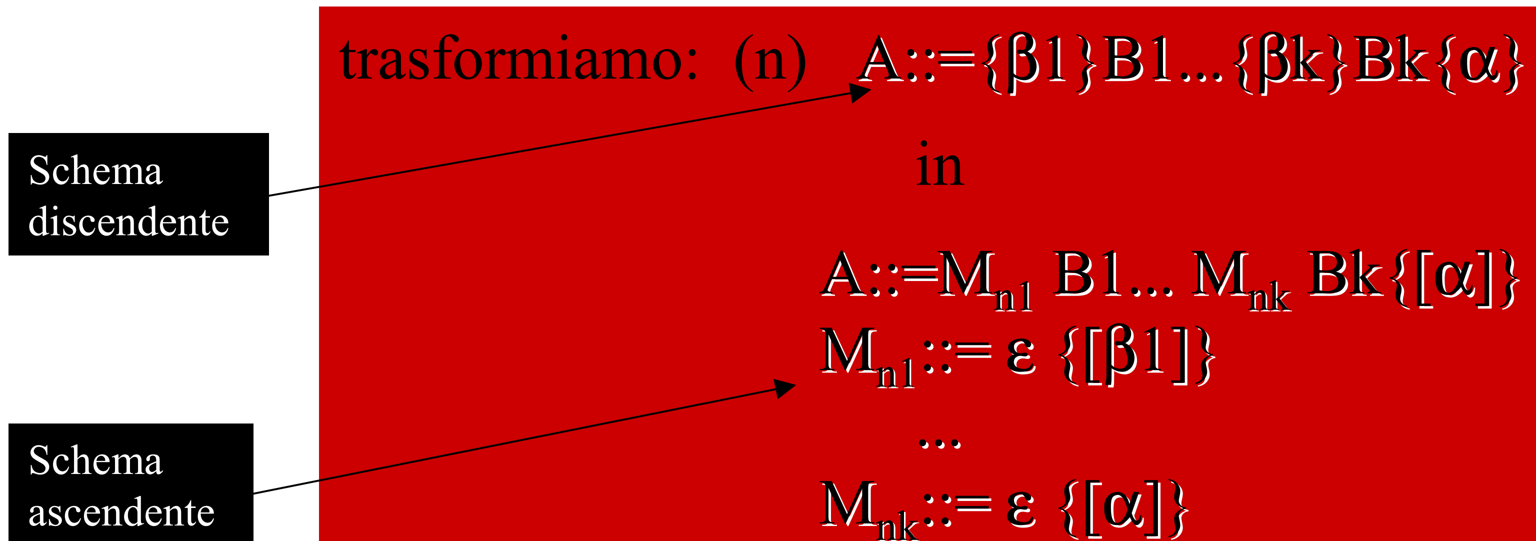
Tutti gli attributi per calcolare α



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

Bottom-up: Trasformazioni per L-attributate

MARCATORI



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*

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