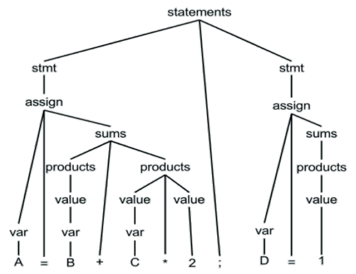
Bottom-up parsing

Fabrizio d'Amore Alberto Marchetti-Spaccamela

DIAG - Sapienza

approcci al parsing



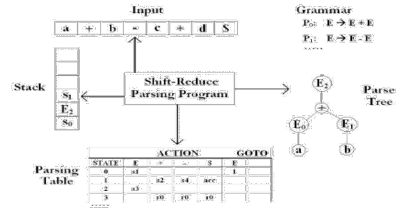
top-down

- **obiettivo**: fissata una grammatica non-contestuale G, data una stringa x in input, costruire un albero di derivazione di x a partire dalla radice, usando le produzioni di G
  - G è fissata
- **metodi**
  - analisi a discesa ricorsiva (esponenziale)
  - analisi predittiva (lineare, basata su grammatiche LL(k))

bottom-up

- **obiettivo**: fissata una grammatica non-contestuale G, data una stringa x in input, costruire un albero di derivazione di x a partire dalle foglie, usando le produzioni di G
  - G è fissata
- **metodo**: analisi ***shift-reduce***
  - lineare
  - analisi predittiva(basata su grammatiche LR(k))
  - esistono altri metodi

parsing shift-reduce

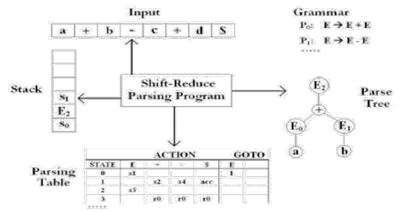


L'idea generale

- leggere i token dall'input e inserirli in stack tentando di costruire sequenze che riconosciamo come il lato destro di una produzione
- Quando si trova la corrispondenza, sostituire quella sequenza con il non terminale dal lato sinistro
- Questo processo crea l'albero di analisi dal fondo verso l'alto (bottom-up), l'inverso del parser top-down.
- Se tutto va bene, alla fine si esamina tutto l'input e in stack rimane il simbolo iniziale

- Si eseguono "sostituzioni inverse" (riduzioni) su sottostringhe dell'input, percorrendo al contrario il processo di derivazione
- Utilizza una pila (inizialmente vuota), e si legge l'input un token alla volta,
- Operazioni possibili ad ogni lettura
  - **shift** (continua a leggere input e push stack)
  - **Reduce** (applica una produzione e riduci pila)
  - se non è possibile operare shift o reduce allora o **errore** oppure termina e si accetta la stringa

parsing shift-reduce



- ricostruisce una derivazione sinistra della stringa in input, o restituisce messaggio di errore in caso la derivazione non esista
- durante l'esecuzione effettua "sostituzioni inverse" (riduzioni) su sottostringhe dell'input, percorrendo al contrario il processo di derivazione
- basato su una pila (inizialmente vuota), legge l'input un token alla volta, eseguendo ad ogni lettura uno **shift** oppure uno o più **reduce**
  - se non è possibile operare uno shift o una reduce allora si produce un messaggio di **errore**

l'input è diviso in due parti quella ancora da leggere (undigested) e quella letta, inserita in pila e parzialmente processata (semi-digested)

- operazione **shift**
  - *sposta in pila* (shift) prossimo token in input
- operazione **reduce**
  - individua stringa $\alpha \in V^*$ affiorante nella pila, tale che esiste $X \rightarrow \alpha$, *esegui pop di tutti i caratteri di α ed esegui push di X* (reduce); α è detta **handle**
  - dopo una reduce potrebbe essere possibile eseguire un'altra reduce
  - se dopo una reduce la pila contiene solo l'assioma e l'input è stato letto interamente allora il parsing termina con **successo**

semplice esempio

linguaggio non-contestuale
$L = \{a^n b^m c^{n+m} \mid n, m > 0\}$

Produzioni (S assioma, simb. terminali a,b,c))

$S \rightarrow aSc \mid aTc$

$T \rightarrow bTc \mid bc$

esempio generazione *abbbcccc*

$\underline{S} \Rightarrow a\underline{T}c \Rightarrow ab\underline{T}cc \Rightarrow abb\underline{T}ccc \Rightarrow abbbcccc$

HANDLE: sequenza di caratteri in pila che sono usati in reduce; sono il lato destro della produzione (in rosso e sottolineato a destra)

simbolo convenzionale di fine input: $

	pila	undigested	azione
1		*abbbcccc$*	shift
2	*a*	*bbbcccc$*	shift
3	*ab*	*bbcccc$*	shift
4	*abb*	*bcccc$*	shift
5	*abbb*	*cccc$*	shift
6	$abb\underline{bc}$	*ccc$*	reduce ($T \rightarrow bc$)
7	*abbT*	*ccc$*	shift
8	$ab\underline{bTc}$	*cc$*	reduce ($T \rightarrow bTc$)
9	*abT*	*cc$*	shift
10	$a\underline{bTc}$	*c$*	reduce ($T \rightarrow bTc$)
11	*aT*	*c$*	shift
12	$\underline{aTc}$	$	reduce ($S \rightarrow aTc$)
13	*S*	$	accetta

Esempio

data la grammatica S → E E → T | E + T T → id | (E)
e la stringa in input (id+id)

PARSE STACK	REMAINING INPUT	PARSER ACTION
	(id + id)$	Shift (push next token from input on stack, advance input)
(	id + id)$	Shift
(id	+ id)$	Reduce: T –> id (pop right-hand side of production off stack, push left-hand side, no change in input)
(T	+ id)$	Reduce: E –> T
(E	+ id)$	Shift
(E +	id)$	Shift
(E + id	)$	Reduce: T –> id
(E + T	)$	Reduce: E –> E + T (Ignore: E –> T)
(E	)$	Shift
(E)	$	Reduce: T –> (E)
T	$	Reduce: E –> T
E	$	Reduce: S –> E
S	$	

Passi
1: stack vuoto; input da esaminare: tutto; azione: shift (poni primo input in stack)
2: shift
3: in input id; si applica (reduce) T→id equivale a togliere id da stack e mettere al suo posto T; input non cambia
…
Shift: avanza in input
Reduce: applica produzione e togli uno o piu' elementi da stack (lato destro prod. e li sostituisci con un solo simbolo (lato sinistro produzione)
Alla fine: stack e input hanno solo S (simbolo iniziale – stack) e $ (simbolo fine input)

Esempio data la grammatica $S \rightarrow E \quad E \rightarrow T \mid E + T \quad T \rightarrow id \mid (E)$

stringa input (id+id)

PARSE STACK	REMAINING INPUT	PARSER ACTION
	(id + id)$	Shift (push next token from input on stack, advance input)
(	id + id)$	Shift
(id	+ id)$	Reduce: T –> id (pop right-hand side of production off stack, push left-hand side, no change in input)
(T	+ id)$	Reduce: E –> T
(E	+ id)$	Shift
(E +	id)$	Shift
(E + id	)$	Reduce: T –> id
(E + T	)$	Reduce: E –> E + T (Ignore: E –> T)
(E	)$	Shift
(E)	$	Reduce: T –> (E)
T	$	Reduce: E –> T
E	$	Reduce: S –> E
S	$	

Conflitti: Nell'analisi precedente potevamo ridurre in due modi diversi

In particolare, abbiamo ignorato la possibilità di ridurre $E \rightarrow T$ perché ciò avrebbe creato la sequenza (E + E nello stack che non è un prefisso praticabile di una forma sentenziale corretta.

Infatti non esiste un lato destro che corrisponda alla sequenza (E + E e nessuna possibile riduzione che la trasforma in tale, questo è un vicolo cieco e non viene considerato.

Vedremo come il parser può determinare quali riduzioni sono valide in una situazione particolare.

conflitti

ad ogni passo del parsing potrebbero nascere **conflitti**

- *reduce-reduce*, ovvero potrei eseguire il *reduce* con due produzioni differenti:
  - come scegliere la produzione giusta?
- *shift-reduce*, ovvero potrei eseguire *shift* o *reduce*
  - Come scegliere fra shift o reduce?
  - esempio del *dangling else (else* "appeso")

Dangling else (else appeso):
conflitto *shift-reduce*

`S → if E then S | if E then S else S`

se il parser esaminando la sequenza
`if E then if E then S else S`
avremo a un certo punto nella pila
`if E then if E then S`

il prossimo token in input è `else`

- *reduce* cambia la pila in
  `if E then S`
- *shift* (e successivi *shift*) cambia la pila in
  `if E then if E then S else S`

le due possibilità hanno diverse conseguenze nell'attribuire l'`else` (al primo o al secondo `if`, rispettivamente nei casi *reduce* e *shift*)

Quale opzione è quella usata nei linguaggi che conoscete?

conflitti

- Se applico 1 ottengo

  if E then {if E then S} else S
- Se applico 2 ottengo

  if E then {if E then S else S}

L'ambiguità viene risolta con

- una regola aggiuntiva (che non appare nella grammatica): ad es. in C, Java si sceglie interpretazione 2
- Obbligando ad usare parentesi {} per delimitare (ad es. ADA, Modula)

NOTA i conflitti *reduce-reduce* sono rari e sono causati da problemi nella costruzione delle grammatiche

Dangling else (else appeso):
conflitto *shift-reduce*

`S → if E then S | if E then S else S`

se il parser sta derivando

`if E then if E then S else S`

avremo a un certo punto nella pila

`if E then if E then S`

il prossimo token in input sarà `else`

1. *Applico reduce* cambia la pila in

   `if E then S`

2. *shift* (e successivi *shift*) cambia la pila in

`if E then if E then S else S`

le due possibilità hanno diverse conseguenze nell'attribuire l'`else` (al primo o al secondo `if`, rispettivamente nei casi *reduce* e *shift*)

l'ambiguità viene risolta con una regola aggiuntiva che non appare nella grammatica

i conflitti *reduce-reduce* sono rari e sono causati da problemi nella costruzione delle grammatiche

conflitti

Conflitti *reduce-reduce*

Poter eseguire *reduce* con due produzioni differenti

Esempio:
$L' = \{a^n b^m c^{n+m} \mid n+m > 0\}$
$= L \cup \{ac, bc\}$
produzioni
$S \rightarrow aSc \mid aTc \mid ac \mid bc$
$T \rightarrow bTc \mid bc$

generazione *abbbcccc*
$\underline{S} \Rightarrow a\underline{T}c \Rightarrow ab\underline{T}cc \Rightarrow abb\underline{T}ccc \Rightarrow$
$abbbcccc$

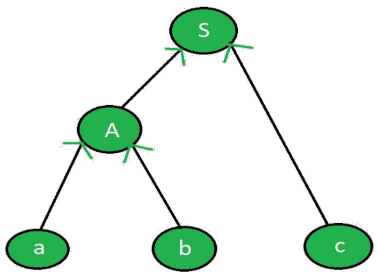
Nota: questi conflitti si possono risolvere con il parser bottom-up

simbolo convenzionale di fine input: $

passi	pila	undigested	azione
1		*abbbcccc$*	shift
2	*a*	*bbbcccc$*	shift
3	*ab*	*bbcccc$*	shift
4	*abb*	*bcccc$*	shift
5	*abbb*	*cccc$*	shift
6	$abb\underline{bc}$	*ccc$*	reduce ($T \rightarrow bc$)
7	*abbT*	*ccc$*	shift
8	$ab\underline{bTc}$	*cc$*	reduce ($T \rightarrow bTc$)
9	*abT*	*cc$*	shift
10	$a\underline{bTc}$	*c$*	reduce ($T \rightarrow bTc$)
11	*aT*	*c$*	shift
12	$\underline{aTc}$	$	reduce ($S \rightarrow aTc$)
13	*S*	$	end

perché non $S \rightarrow bc$?

LR parsing (Left Right)



- per risolvere i conflitti occorre stabilire come
  - riconoscere un *handle*
  - decidere la produzione da usare in una riduzione (in presenza di conflitti *reduce/reduce*)
  - utilizzare strumenti (come il *lookahead*) per dirimere un conflitto *shift/reduce*
- studieremo **parser LR**
  - **L**: left-to-right scan dell'input
  - **R**: costruzione di una derivazione rightmost (destra)

Confronto fra parser LR (bottom-up) e parser LL (top-down)

- le grammatiche che ammettono parsing LR sono più numerose di quelle che ammettono parsing predittivo LL
  - in pratica includono tutte quelle dei linguaggi di programmazione
- per il parsing LR c'è un minor bisogno di "aggiustare" le produzioni,
- c'è un lungo lavoro di preparazione di tabelle che permettono di riconoscere casi e definire azioni – per fortuna automatizzabile, poiché non richiede creatività
- Una volta realizzate le tabelle la realizzazione del parser (sia LL che LR) è semplice

LR parsing: diverso uso della pila

Vediamo prima come realizzare il parsing; poi come costruire le tabelle

- nel parsing shift-reduce la pila contiene le forme di frase che vengono via via elaborate per applicare in senso inverso le produzioni, fino ad arrivare all'assioma
- in molti casi la produzione da applicare non dipende solo da quale è lo handle individuato ma anche dagli altri simboli in pila, e quindi da un contesto
- **per meglio catturare il contesto si mettono in pila non semplici token, ma veri e propri stati, che rappresentano il contesto sinistro al momento**
- **lo stato che affiora dalla pila,** eventualmente aiutato da un *lookahead*, consente di prendere la decisione corretta (*shift* o *reduce*, e con quale produzione)
  - una più precisa definizione di stato verrà data nel seguito attraverso opportuni automi a stati finiti

tavole Action e Goto

i parser LR fanno uso di due tavole

Action table Action[*s*, *a*] Goto table Goto[*s*, *X*]

State	Goto		Action				
	E	T	id	(	)	+	$
0	G1	G8	S4				
1						S2	ACC
2		G3	S4				
3					R2	R2	R2
4				S5	R5	R5	R5
5	G6	G8	S4				
6					S7	S2	
7					R4	R4	R4
8					R3	R3	R3

Action[*s*, *a*] (Azioni)

descrive quale azione eseguire quando lo stato affiorante in pila è *s* e il prossimo token in input è il terminale *a*

Possibili azioni

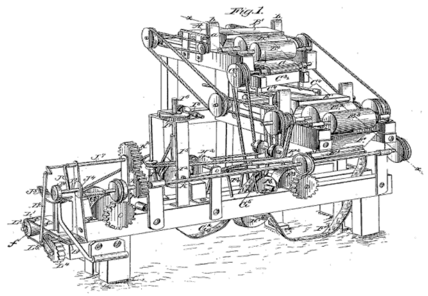
- ***Shift***: inserire uno stato sulla pila (push)
- ***Reduce***: uno handle associato a uno o più stati in cima alla pila (attenzione: si eliminano solo elementi dalla pila)
- ***accept***, termina con successo
- ***report error***, se non è possibile procedere

Goto[*s*, *X*]

indica il nuovo stato da piazzare in cima alla pila (push) dopo la riduzione del non-terminale *X*, mentre lo stato affiorante è *s*, e serve per completare la riduzione

- riduzione del non-terminale *X* significa che è stato individuato uno handle α ed usata la produzione $X \rightarrow \alpha$
- Dopo aver eliminato da stack i simboli α affiora in pila lo stato *s*
- l'indicazione della tavola Goto è il nuovo stato di cui fare il push (dopo i pop già eseguiti)

operazioni di un LR parser



pila inizializzata con stato iniziale s_0 token in input: a stato corrente: s_t

Parser usa due tavole

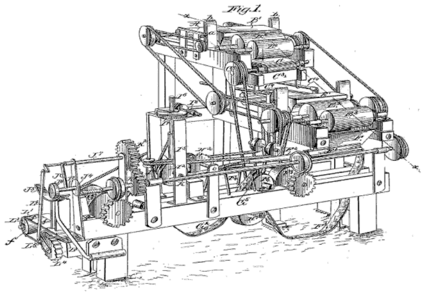
Action specifica se eseguire shift o reduce e i dettagli

GOTO indica il nuovo stato da inserire in cima allo stack dopo una riduzione

L'azione del parser è la seguente:

- Se Action [s, a] è shift st, inseriamo lo stato t nello stack. Quindi si esamina il token successivo dall'input.
- Se Action [st, a] è reduce Y -> X1 ... Xk, elimina k stati X1, …Xk dallo stack lasciando lo stato su in alto. GOTO[s_u, Y] dà il nuovo stato da inserire sullo stack. Il token di input è lo stesso (ovvero, l'input rimane invariato).
- Se Action [st, a] è ACCETTA, l'analisi ha esito positivo e abbiamo terminato.
- Se Action [st, a] è ERRORE (ovvero la posizione della tabella è vuota), abbiamo un errore di sintassi.

operazioni di un LR parser



pila inizializzata con stato iniziale s_0

token in input: a stato corrente (in cima alla pila): s_t

- **Action[s_t, a] == shift(s_u)**

  esegue push(s_u) e legge il prossimo token

- **Action[s_t, a] == reduce($A \rightarrow B_1 \ldots B_k$)**

  esegue k pop (estrazione di k stati), dopodiché, se s_v è lo stato affiorante, esegue push(**GOTO(s_v , A)**); il token in input non cambia

- **Action[s_t, a] == accept**

  il parser conclude le operazioni con successo

- **Action[s_t, a] == error**

  errore sintattico (con la pila corrente e il token in input non è possibile raggiungere una forma di frase con un handle da ridurre)
  in questo caso è in genere possibile stampare messaggio diagnostico sull'errore

N.B. per ciascuna reduce si fornisce in output la produzione ridotta

LR parsing in azione

Grammatica

1) E –> E + T   2) E –> T   3) T –> (E)   4) T –> id

Tavole Action e GOTO unite

(sy indica shift e nuovo stato y - rx indica reduce su produz. x)

Input : id+(id)

Stato in cima stack	Action					GOTO	
	id	+	(	)	$	E	T
0	s4		s3			1	2
1		s5			accetta		
2	r2	r2	r2	r2	r2		
3	s4		s3			6	2
4	r4	r4	r4	r4	r4		
5	s4		s3				8
6		s5		s7			
7	r3	r3	r3	r3	r3		
8	r1	r1	r1	r1	r1		

PILA	INPUT	Azioni
s0	id + (id)$	Shift: s4 in pila; avanza in input
s0 s4	+ (id)$	Reduce: 4) T –> id; Pop da pila; goto(s0,T)= 2 inserisci s2 in pila input non cambia
s0 s2	+ (id)$	Reduce: 2) E –> T; goto s1
s0 s1	+ (id)$	Shift: s5 in pila; avanza in input

Esempio

Grammatica 1) E –> E + T 2) E –> T 3) T –> (E) 4) T –> id

Input id+(id)

Tavole Action e GOTO insieme

Stato in cima stack	Action					GOTO	
	id	+	(	)	$	E	T
0	s4		s3			1	2
1		s2			accetta		
2	r2	r2	r2	r2	r2		
3	s4		s3			6	2
4	r4	r4	r4	r4	r4		
5	s4		s3				8
6		s5		s7			
7	r3	r3	r3	r3	r3		
8	r1	r1	r1	r1	r1		

analisi di id+(id (continua))

PILA	INPUT	Azioni
s0 s1 s5	(id)$	Shift: s3
s0 s1 s5 s3	id)$	Shift s4
s0s1s5s3 s4	)$	Reduce: 4) T –>id; goto s2
s0s1s5s3s2	)$	Reduce: 2) E –>T; goto s6
s0s1s5s3s6	)$	Shift s7
s0s1s5s3s6s7	$	Reduce 3: T→(E) goto s8
s0s1s5 s8	$	Reduce 1: E→E+T goto s1
s0 s1	$	accetta

Esempio
Grammatica 1) E –> E + T   2) E –> T   3) T –> (E)   4) T –> id
Input id+(id)

Tavole Action e GOTO insieme

Stato in cima stack	Action					GOTO	
	id	+	(	)	$	E	T
0	s4		s3			1	2
1		s2			accetta		
2	r2	r2	r2	r2	r2		

Nota
Le produzioni 3 e 1 nel lato destro hanno tre simboli;
Quindi si eliminano tre stati dalla pila e si inserisce uno stato (dato da goto) che corrisponde al lato sin. della produzione
Dopo eliminazione s0 affiora in pila
Dato che GOTO(s0,E)=s1, metto in pila s1

analisi di id+(id (continua))

PILA	INPUT	Azioni
s0 s1 s5	(id)$	Shift: s3
s0 s1 s5 s3	id)$	Shift s4
s0s1s5s3s4	)$	Reduce: 4) T –>id; goto s2
s0s1s5s3s2	)$	Reduce: 2) E –>T; goto s6
s0s1s5s3s6	)$	Shift s7
s0s1s5s3s6s7	$	Reduce 3: T→(E) goto s8
s0s1s5 s8	$	Reduce 1: E→E+T goto s1
s0 s1	$	accetta

LR parsing in azione

L' = $\{a^n b^m c^{n+m} \mid n+m > 0\}$

G': $S' \rightarrow S$
$S \rightarrow aSc \mid aTc \mid ac \mid bc$
$T \rightarrow bTc \mid bc$

Input: *abbbcccc* S'→S→aTc→ abTcc→abbTccc→ abbbcccc

Pila

- <s0>, input a: shift s2
- <s0 s2>, input b:shift s7
- <s0 s2 s7>, input b: shift s13
- <s0 s2 s7 s13> input b:shift s13
- <s0 s2 s7 s13 s13> input c:shift 15
- <s0 s2 s7 s13 s13 s15> input c: reduce T→bc (togli s13 e s15 da pila; ora in pila affiora s13; vedi che goto (13, T)= 12; metti s12 in pila)

stato	*a*	*b*	*c*	*$*	*S*	*T*
0	*shift* 2	*shift* 3			*goto* 1	
1				*accept*		
2	*shift* 2	*shift* 7	*shift* 6		*goto* 4	*goto* 5
3			*shift* 8			
4			*shift* 9			
5			*shift* 10			
6	*reduce* $S \rightarrow ac$					
7		*shift* 13	*shift* 11			*goto* 12
8	*reduce* $S \rightarrow bc$					
9	*reduce* $S \rightarrow aSc$					
10	*reduce* $S \rightarrow aTc$					
11	*reduce* $S \rightarrow bc$ *reduce* $T \rightarrow bc$					
12			*shift* 14			
13		*shift* 13	*shift* 15			*goto* 12
14	*reduce* $T \rightarrow bTc$					
15	*reduce* $T \rightarrow bc$					
	Tavola Action				**Tavola Goto**	

LR parsing in azione

L' = $\{a^n b^m c^{n+m} \mid n,m > 0\}$
= L ∪ $\{ac, bc\}$

G': $S' \rightarrow S$
$S \rightarrow aSc \mid aTc \mid ac \mid bc$
$T \rightarrow bTc \mid bc$

Input: *abbbcccc* S'→S→aTc→
abTcc→abbTccc→ abbbcccc

Pila

- <s0>, input a: shift s2
- <s0 s2>, input b:shift s7
- <s0 s2 s7>, input b: shift s13
- <s0 s2 s7 s13> input b:shift s13
- <s0 s2 s7 s13 s13> input c:shift 15
- <s0 s2 s7 s13 s13 s15> input c: reduce T→bc (togli s13 e s15 da pila, affiora s13 vedi che goto (13, T)= 12; metti s12 in pila)

stato	*a*	*b*	*c*	*$*	*S*	*T*
0	*shift* 2	*shift* 3			*goto* 1	
1				*accept*		
2	*shift* 2	*shift* 7	*shift* 6		*goto* 4	*goto* 5
3			*shift* 8			
4			*shift* 9			
5			*shift* 10			
6	*reduce* $S \rightarrow ac$					
7		*shift* 13	*shift* 11			*goto* 12
8	*reduce* $S \rightarrow bc$					
9	*reduce* $S \rightarrow aSc$					
10	*reduce* $S \rightarrow aTc$					
11	*reduce* $S \rightarrow bc$ *reduce* $T \rightarrow bc$					
12			*shift* 14			
13		*shift* 13	*shift* 15			*goto* 12
14	*reduce* $T \rightarrow bTc$					
15	*reduce* $T \rightarrow bc$					
	Tavola Action				**Tavola Goto**	

LR parsing in azione

- G': $S' \rightarrow S$
  $S \rightarrow aSc \mid aTc \mid ac \mid bc$
  $T \rightarrow bTc \mid bc$
- Analisi di *abbbcccc*
  (*shift* = s, *reduce* = r, *goto* = g)
- s(2), s(7), s(13), s(13), s(15), r($T \rightarrow bc$), g(12), s(14), r($T \rightarrow bTc$), g(12), s(14), r($T \rightarrow bTc$), g(5), s(10), r($S \rightarrow aTc$), g(1)
- in input rimane $ (fine input), per cui: ***accept***

stato	*a*	*b*	*c*	*$*	*S*	*T*
0	*shift* 2	*shift* 3			*goto* 1	
1				*accept*		
2	*shift* 2	*shift* 7	*shift* 6		*goto* 4	*goto* 5
3			*shift* 8			
4			*shift* 9			
5			*shift* 10			
6	*reduce* $S \rightarrow ac$					
7		*shift* 13	*shift* 11			*goto* 12
8	*reduce* $S \rightarrow bc$					
9	*reduce* $S \rightarrow aSc$					
10	*reduce* $S \rightarrow aTc$					
11	*reduce* $S \rightarrow bc$ *reduce* $T \rightarrow bc$					
12			*shift* 14			
13		*shift* 13	*shift* 15			*goto* 12
14	*reduce* $T \rightarrow bTc$					
15	*reduce* $T \rightarrow bc$					
	Tavola Action				**Tavola Goto**	

Parser bottom-up: cosa abbiamo visto finora

- Si definisce handle la sequenza di caratteri più a sinistra che corrisponde al lato destro di una produzione

Idea: dividi input in due parti:

- la parte sinistra (in pila) è l'area di lavoro
- la parte destra è l'input non ancora esaminato

Nota:

- Non si inseriscono direttamente in pila i terminali e i non terminali ma si inseriscono stati
- Gli stati sono utilizzati per risolvere conflitti e memorizzano informazioni su input e su come siamo arrivati a quel punto (come gli stati in ASF)
- poiché applichiamo la prima produzione a sinistra non abbiamo bisogno di tornare indietro

Nel parser bottom-up si esamina input da sinistra a destra.

Ad ogni iterazione si esegue un'operazione di shift o di reduce

- Shift: muovi a destra in input e poni informazioni in pila relative ai terminali esaminati
- Reduce: trovato handle α si applica una produzione del tipo $A \rightarrow \alpha$ eliminando dalla pila il lato destro gli stati corrispondenti a (handle) α e ponendo in pila lo stato corrispondente a non terminale A
- Dove sono gli handle? in cima alla pila

Parser bottom-up: cosa ~~abbiamo visto finora~~ faremo

Nel parser bottom-up si esamina input da sinistra a destra.

Ad ogni iterazione si esegue un'operazione di shift o di reduce

- Dobbiamo capire come costruire le tavole ACTION e GOTO
- Questo richiede essere in grado di riconoscere gli handle (la sequenza di caratteri più a sinistra che corrisponde al lato destro di una produzione)
- Per fare questo dobbiamo riconoscere i conflitti

COSA FAREMO

1. Capire bene le derivazioni destre

   Per fare questo useremo tre punti di vista diversi

2. Data una grammatica costruiremo un automa a stati finiti
3. L'automa a stati finiti ci permette di derivare le tavole ACTION e GOTO

Una prima vista analisi bottom-up: albero sintattico

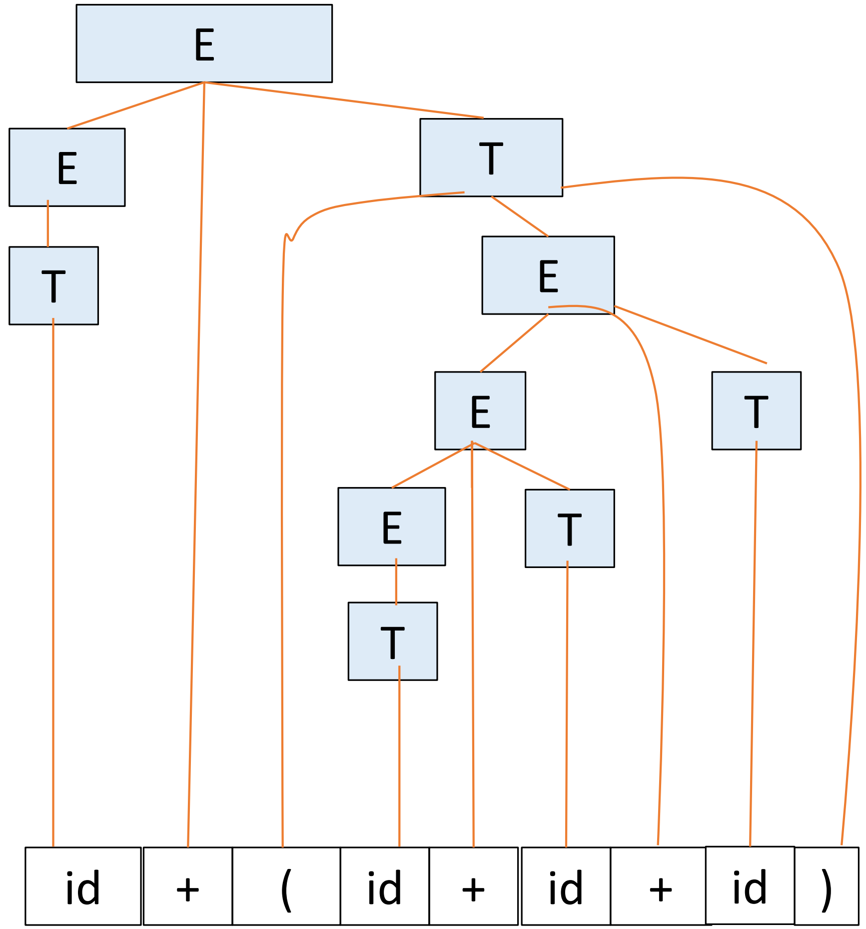
Grammatica

E → T   E → E + T

T → id   T → (E)

Input

id + (id + id + id)



Una seconda vista analisi bottom-up: derivazione destra

$E \rightarrow T \qquad E \rightarrow E + T$

$T \rightarrow id \qquad T \rightarrow (E)$

input id + (id + id + id)

A destra abbiamo la sequenza di derivazioni (reduce) che con l'analisi bottom-up mi porta ad ottenere l'assioma

$id + (id + id + id)$
$\Rightarrow T + (id + id + id)$
$\Rightarrow E + (id + id + id)$
$\Rightarrow E + (T + id + id)$
$\Rightarrow E + (E + id + id)$
$\Rightarrow E + (E + T + id)$
$\Rightarrow E + (E + id)$
$\Rightarrow E + (E + T)$
$\Rightarrow E + (E)$
$\Rightarrow E + T$
$\Rightarrow E$

Una seconda vista analisi bottom-up: derivazione destra

$E \rightarrow T \qquad E \rightarrow E + T$

$T \rightarrow id \qquad T \rightarrow (E)$

L'analisi bottom-up da sinistra a destra (LR) è una derivazione destra tracciata in ordine inverso

Ricorda: in una derivazione destra espando sempre il non terminale più a destra

$$
\begin{aligned}
& id + (id + id + id) \\
\Rightarrow\ & \mathbf{T} + (id + id + id) \\
\Rightarrow\ & \mathbf{E} + (id + id + id) \\
\Rightarrow\ & E + (\mathbf{T} + id + id) \\
\Rightarrow\ & E + (\mathbf{E} + id + id) \\
\Rightarrow\ & E + (E + \mathbf{T} + id) \\
\Rightarrow\ & E + (\mathbf{E} + id) \\
\Rightarrow\ & E + (E + \mathbf{T}) \\
\Rightarrow\ & E + (\mathbf{E}) \\
\Rightarrow\ & E + \mathbf{T} \\
\Rightarrow\ & \mathbf{E}
\end{aligned}
$$

Una terza vista sull'analisi bottom-up

id + (id + id + id)

⇒ T + (id + id + id)

⇒ E + (id + id + id)

⇒ E + (T + id + id)

⇒ E + (E + id + id)

⇒ E + (E + T + id)

⇒ E + (E + id)

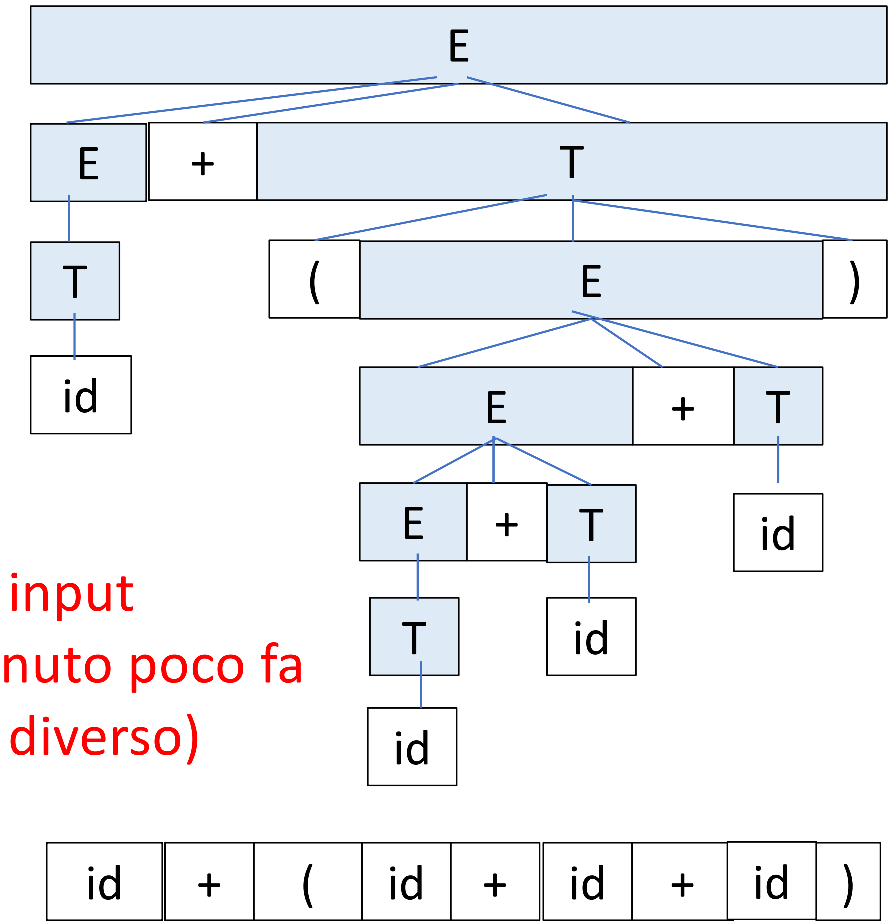
⇒ E + (E + T)

⇒ E + (E)

⇒ E + T

⇒ E

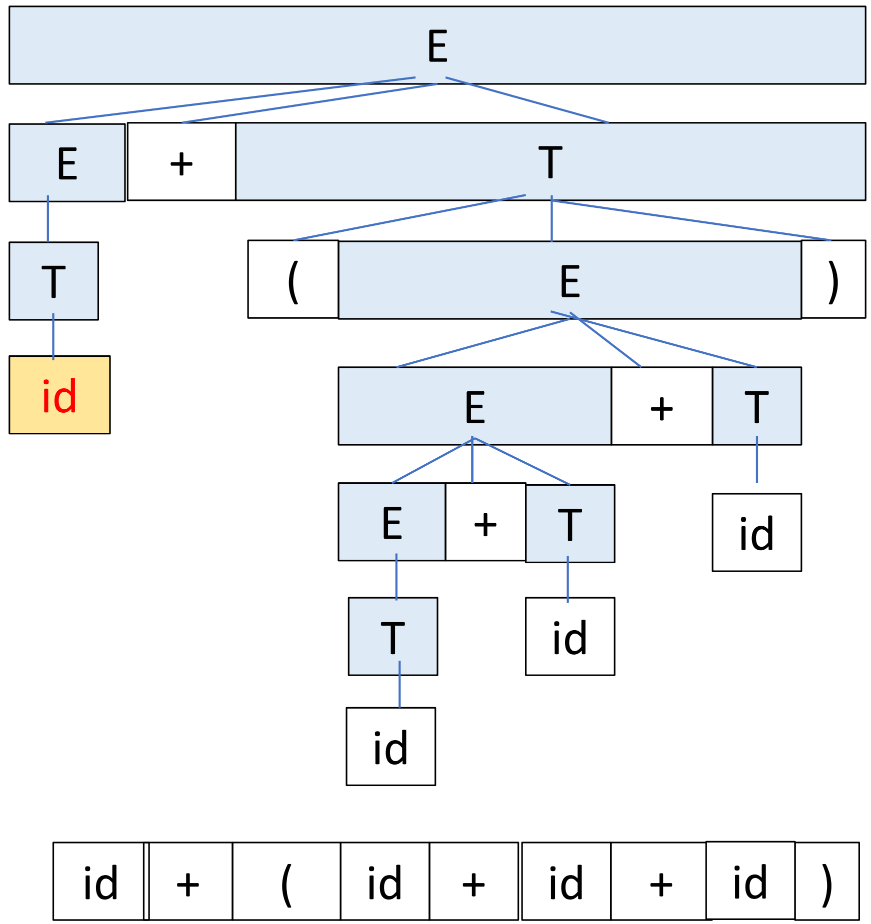
Nota:
le foglie rappresentano input
L'albero è lo stesso ottenuto poco fa
(ma disegnato in modo diverso)



Una terza vista sull'analisi bottom-up

id + (id + id + id)
⇒ T + (id + id + id)
⇒ E + (id + id + id)
⇒ E + (T + id + id)
⇒ E + (E + id + id)
⇒ E + (E + T + id)
⇒ E + (E + id)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

Reduce: Produzione $T \Rightarrow id$ Handle(in rosso) la parte destra della produzione applicata



Una terza vista sull'analisi bottom-up

~~id + (id + id + id)~~

⇒ T + (id + id + id)

⇒ E + (id + id + id)

⇒ E + (T + id + id)

⇒ E + (E + id + id)

⇒ E + (E + T + id)

⇒ E + (E + id)

⇒ E + (E + T)

⇒ E + (E)

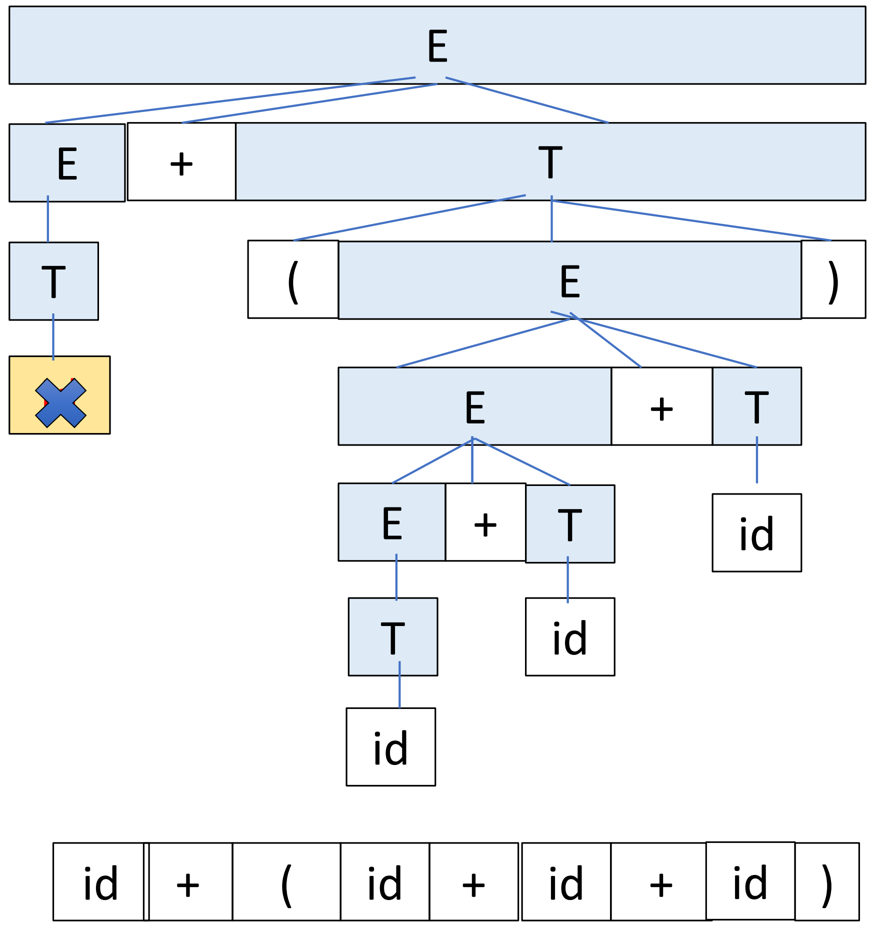
⇒ E + T

⇒ E

Reduce:

Produzione

T ⇒id

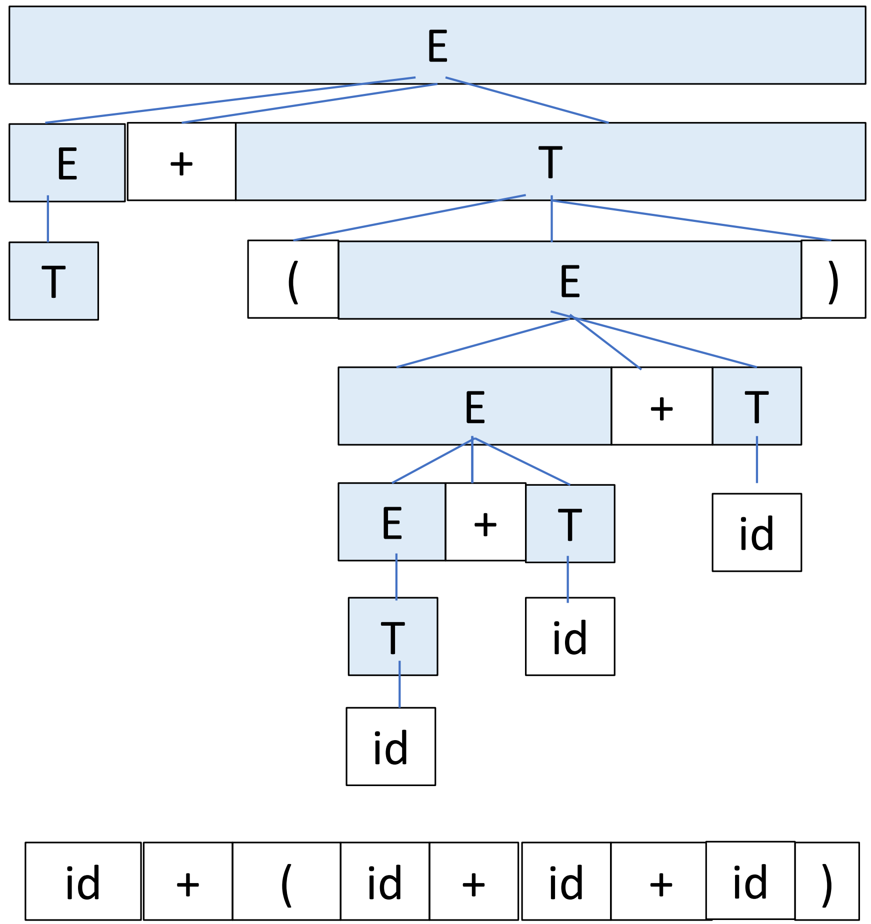


Una terza vista sull'analisi bottom-up

T + (id + id + id)
⇒ E + (id + id + id)
⇒ E + (T + id + id)
⇒ E + (E + id + id)
⇒ E + (E + T + id)
⇒ E + (E + id)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

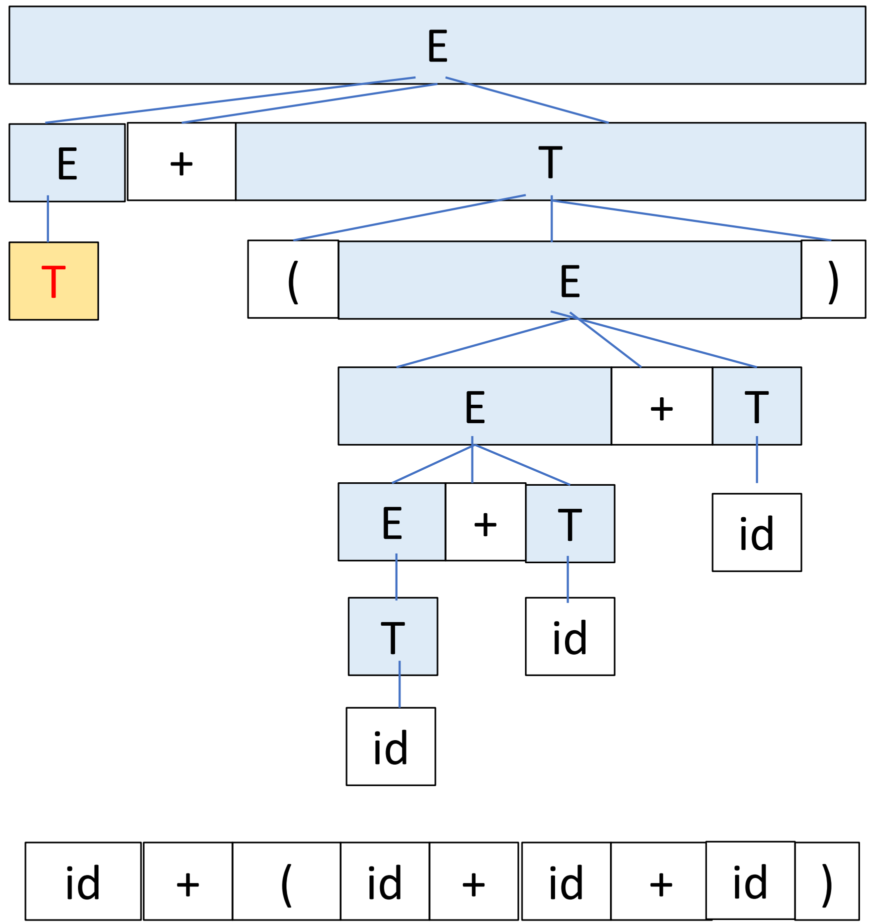
Proseguendo
Otteniamo

Nota: le foglie ora rappresentano la stringa derivata finora dall'assioma



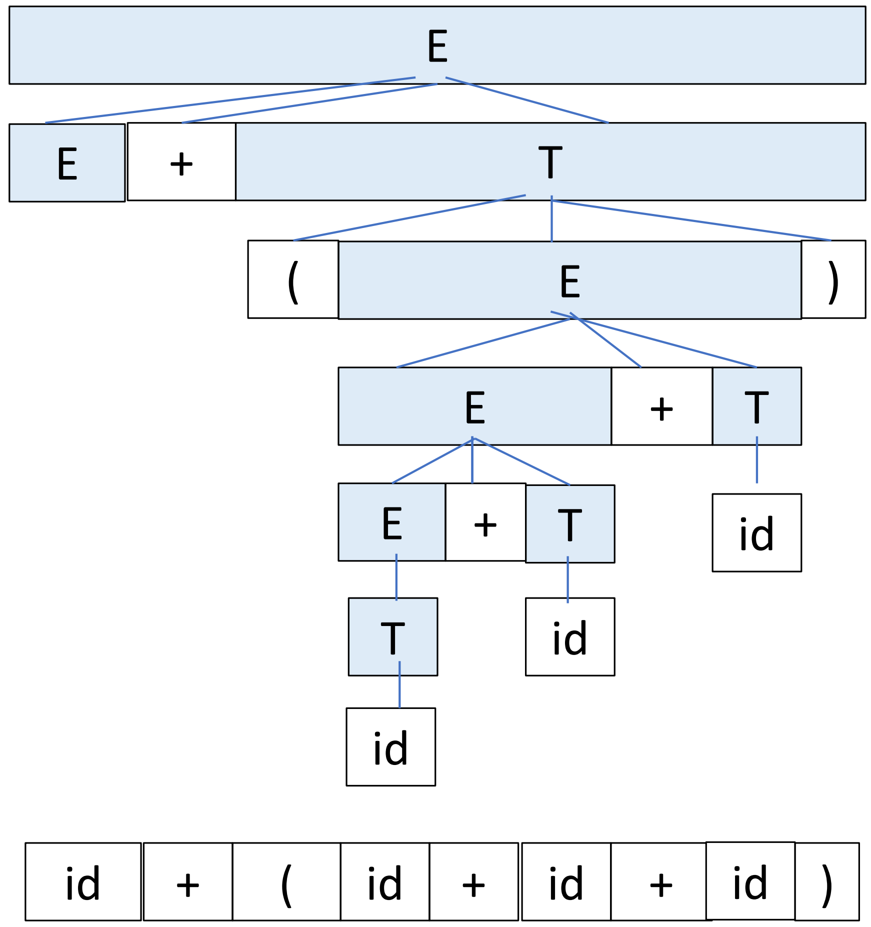
Una terza vista sull'analisi bottom-up

T + (id + id + id)
⇒ E + (id + id + id)
⇒ E + (T + id + id)
⇒ E + (E + id + id)
⇒ E + (E + T + id)
⇒ E + (E + id)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



Una terza vista sull’analisi bottom-up

E + (id + id + id)
⇒ E + (T + id + id)
⇒ E + (E + id + id)
⇒ E + (E + T + id)
⇒ E + (E + id)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



Una terza vista sull'analisi bottom-up

E + (id + id + id)

⇒ E + (T + id + id)

⇒ E + (E + id + id)

⇒ E + (E + T + id)

⇒ E + (E + id)

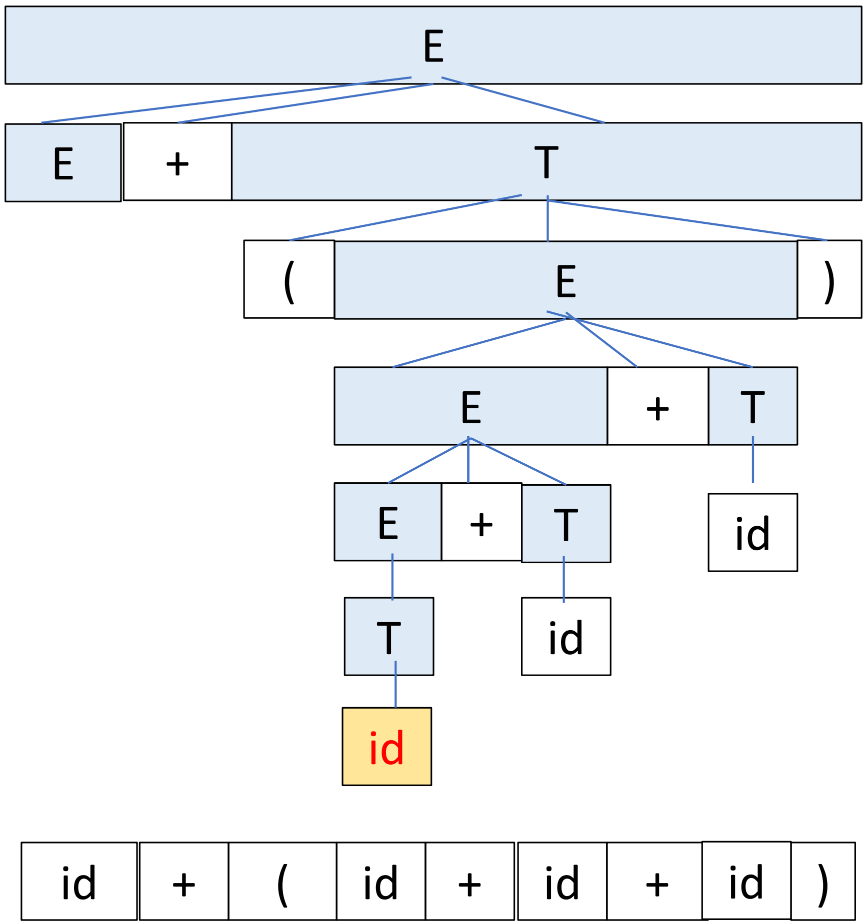
⇒ E + (E + T)

⇒ E + (E)

⇒ E + T

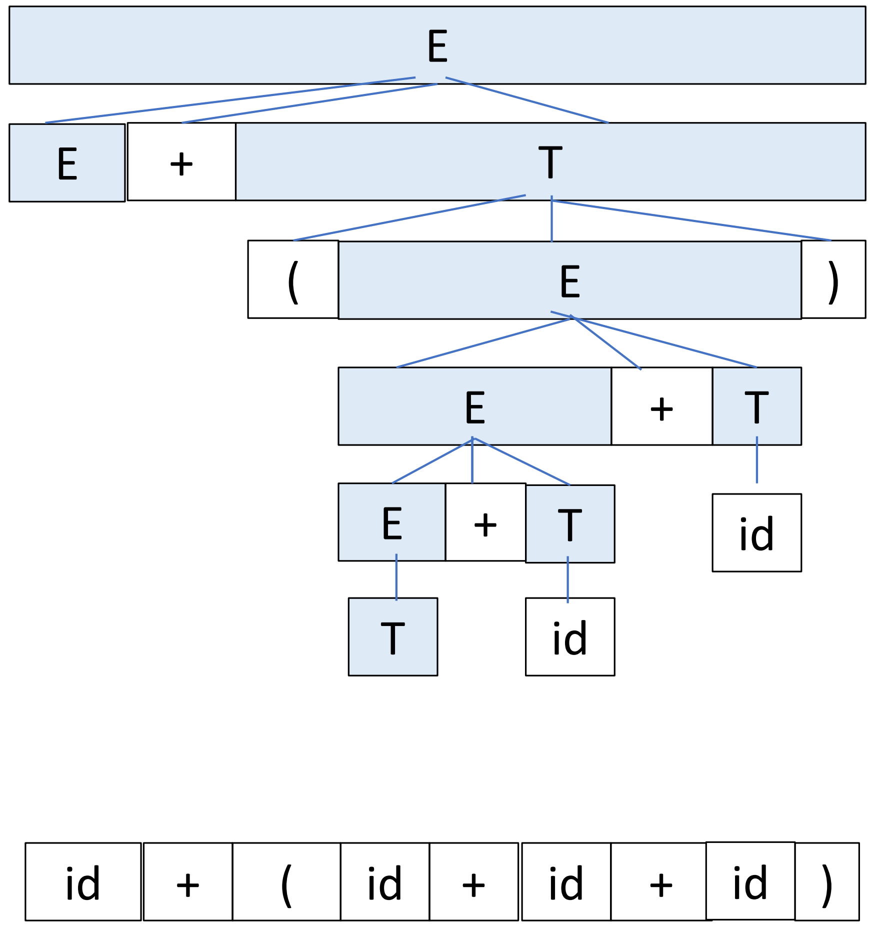
⇒ E

Nota: per eseguire reduce T ⇒id
Bisogna eseguire shift e inserire in pila: + (



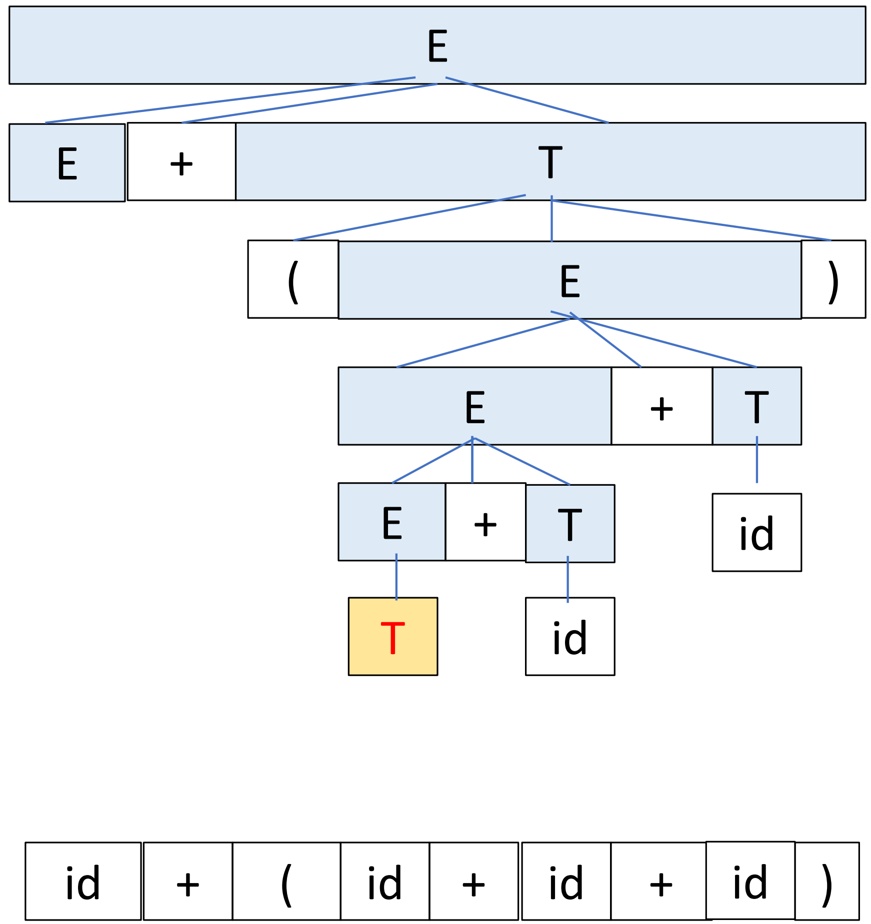
Una terza vista sull'analisi bottom-up

E + (T + id + id)
⇒ E + (E + id + id)
⇒ E + (E + T + id)
⇒ E + (E + id)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



Una terza vista sull'analisi bottom-up

E + (T + id + id)
⇒ E + (E + id + id)
⇒ E + (E + T + id)
⇒ E + (E + id)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



Una terza vista sull'analisi bottom-up

⇒ E + (E + id + id)

⇒ E + (E + T + id)

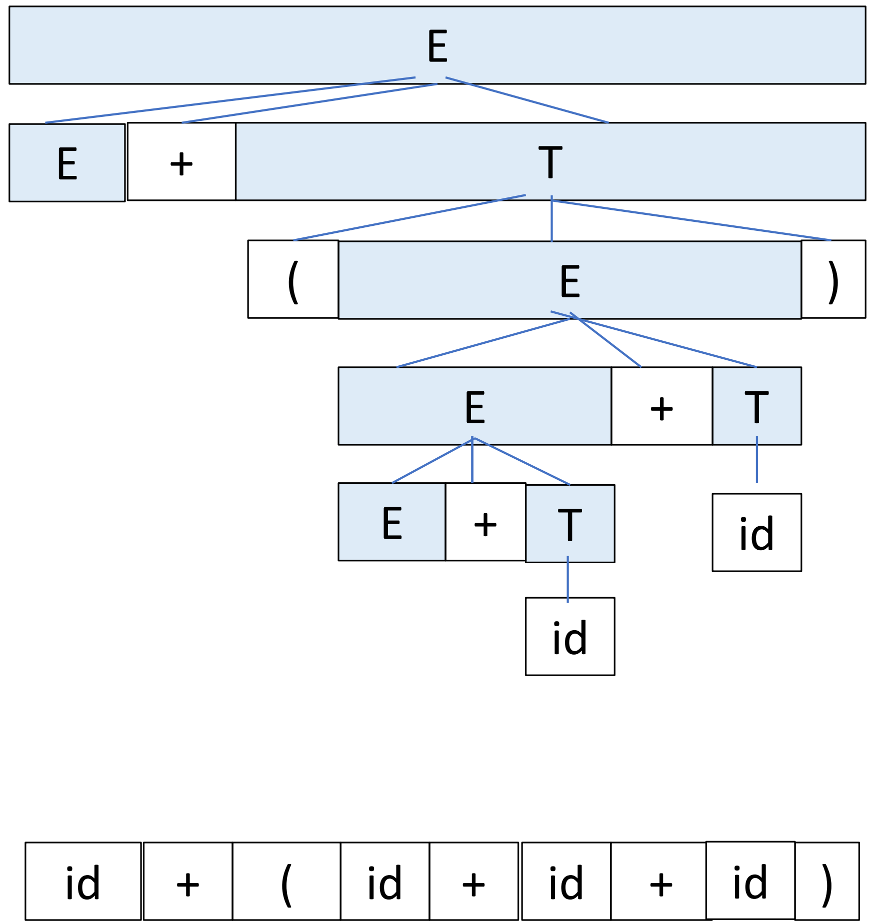
⇒ E + (E + id)

⇒ E + (E + T)

⇒ E + (E)

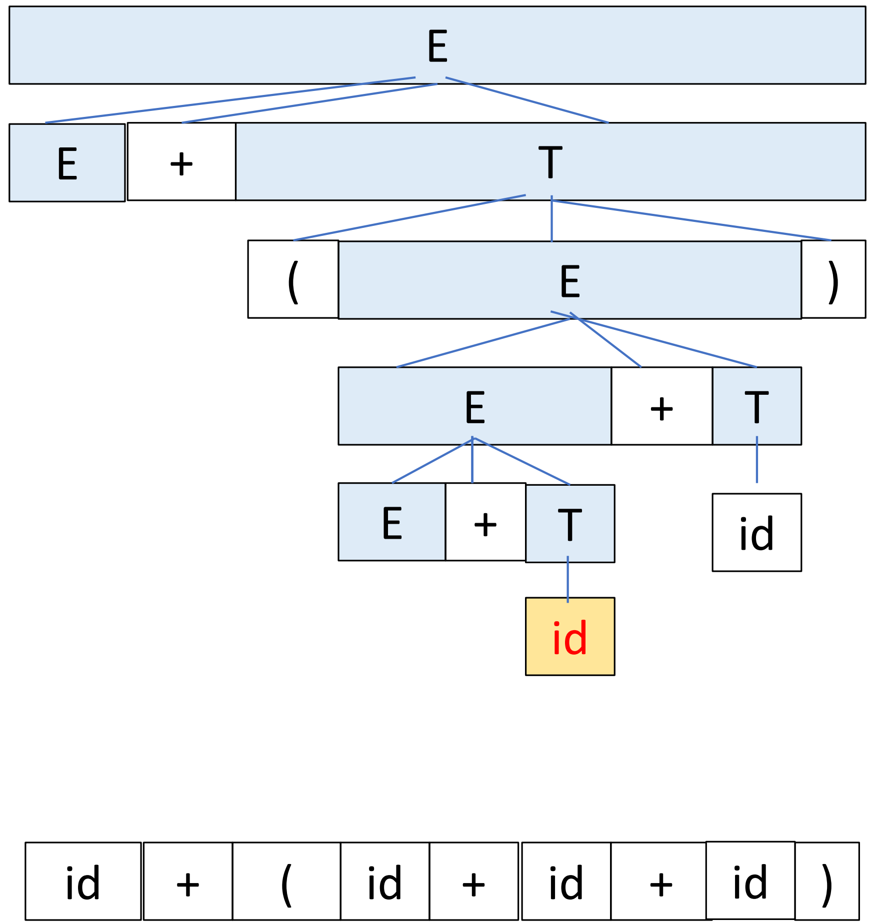
⇒ E + T

⇒ E



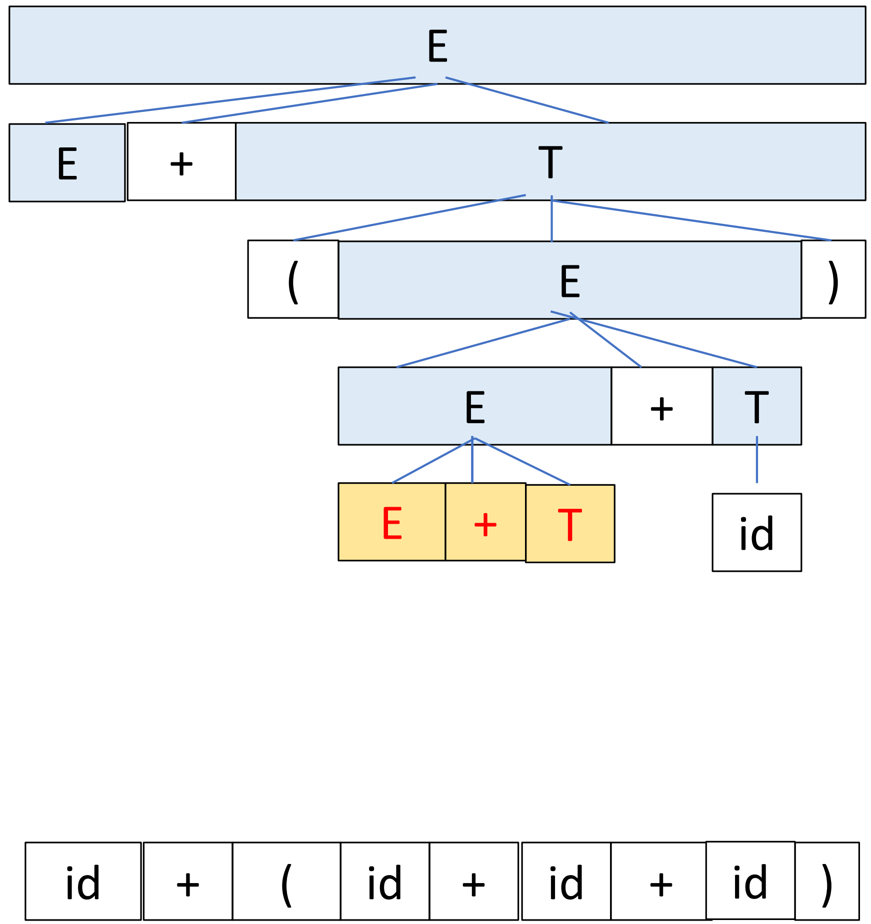
Una terza vista sull'analisi bottom-up

⇒ E + (E + id + id)
⇒ E + (E + T + id)
⇒ E + (E + id)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E



Una terza vista sull'analisi bottom-up

$\Rightarrow$ E + (E + T + id)
$\Rightarrow$ E + (E + id)
$\Rightarrow$ E + (E + T)
$\Rightarrow$ E + (E)
$\Rightarrow$ E + T
$\Rightarrow$ E



Una terza vista sull'analisi bottom-up

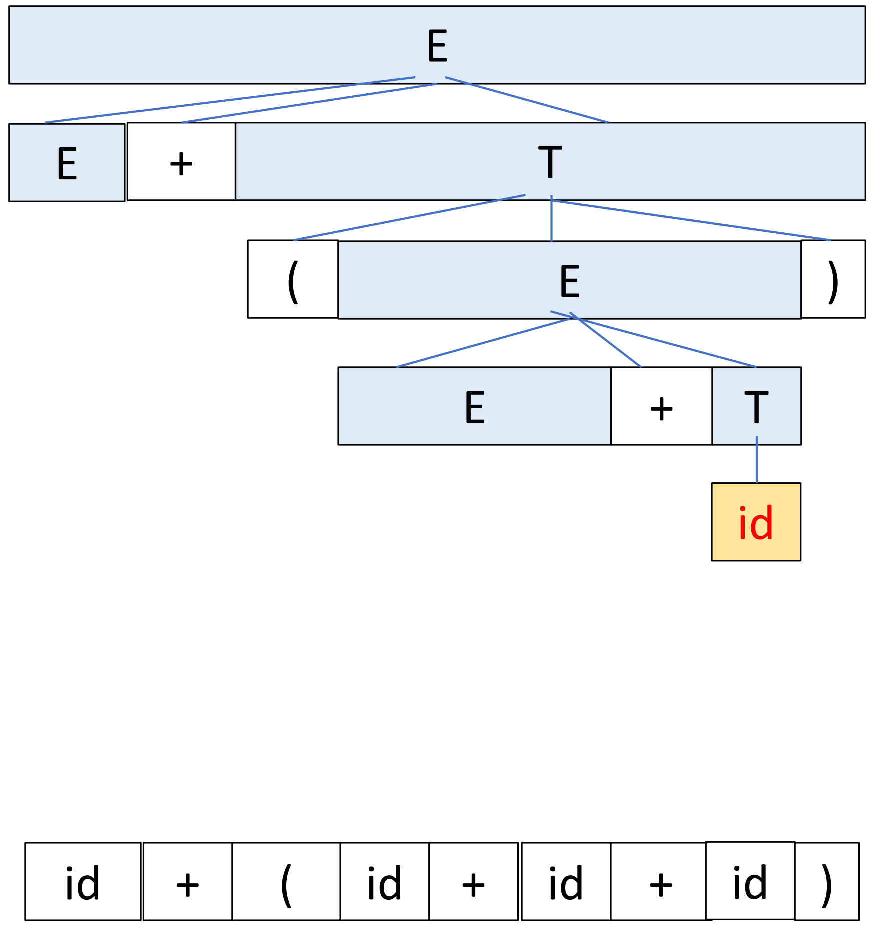
⇒ E + (E + id)

⇒ E + (E + T)

⇒ E + (E)

⇒ E + T

⇒ E



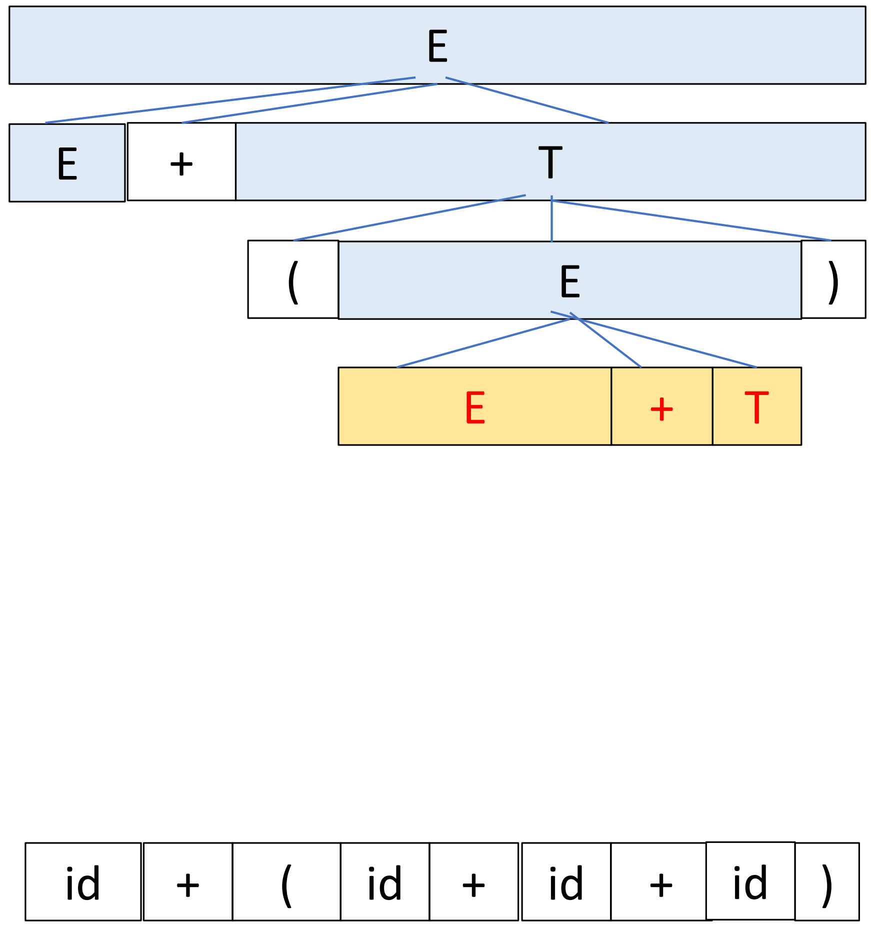
Una terza vista sull'analisi bottom-up

E + (E + T)

⇒ E + (E)

⇒ E + T

⇒ E

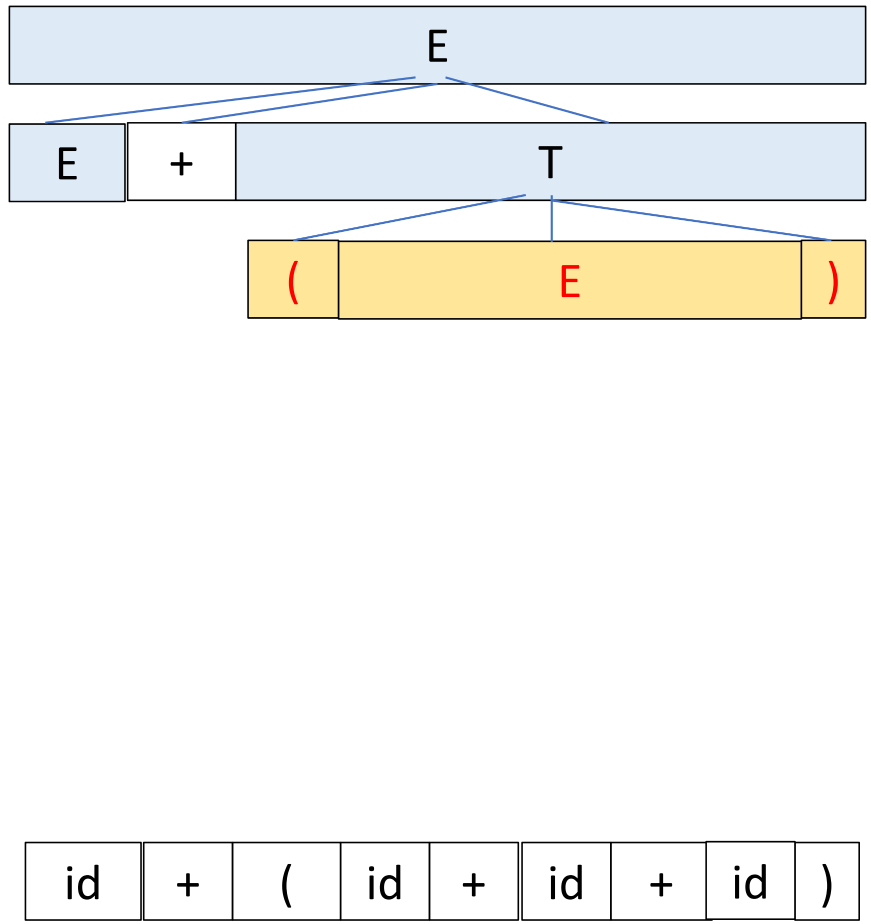


Una terza vista sull'analisi bottom-up

E + (E)

⇒ E + T

⇒ E

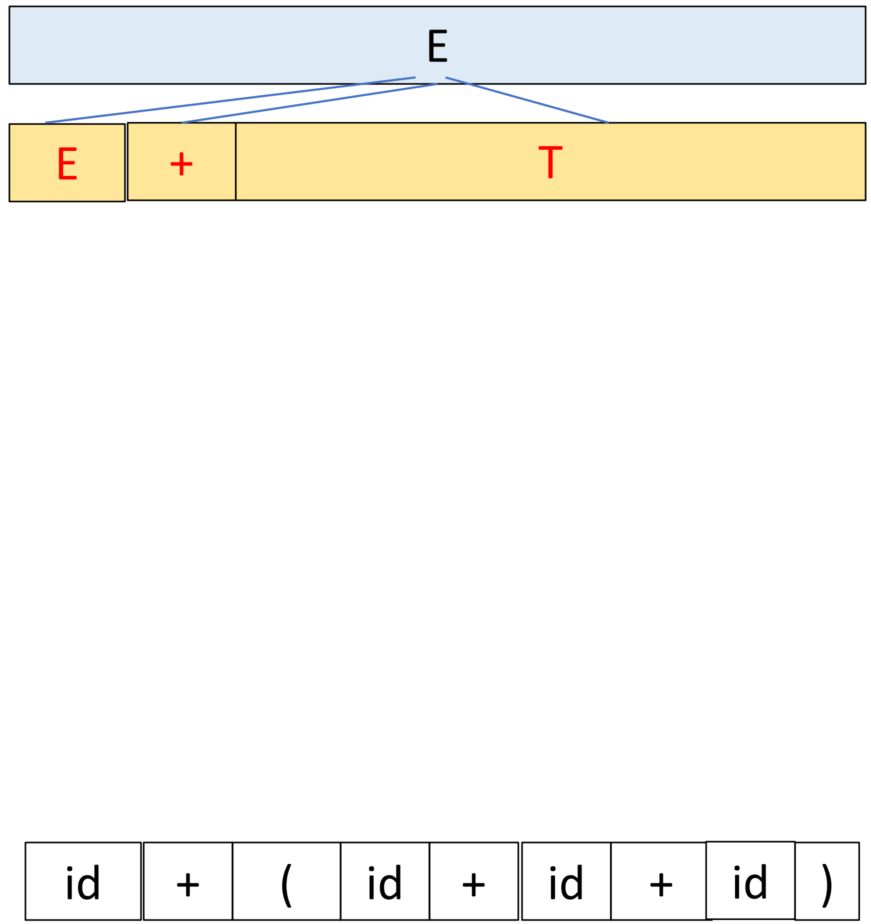


Una terza vista sull'analisi bottom-up

E + T

⇒ E

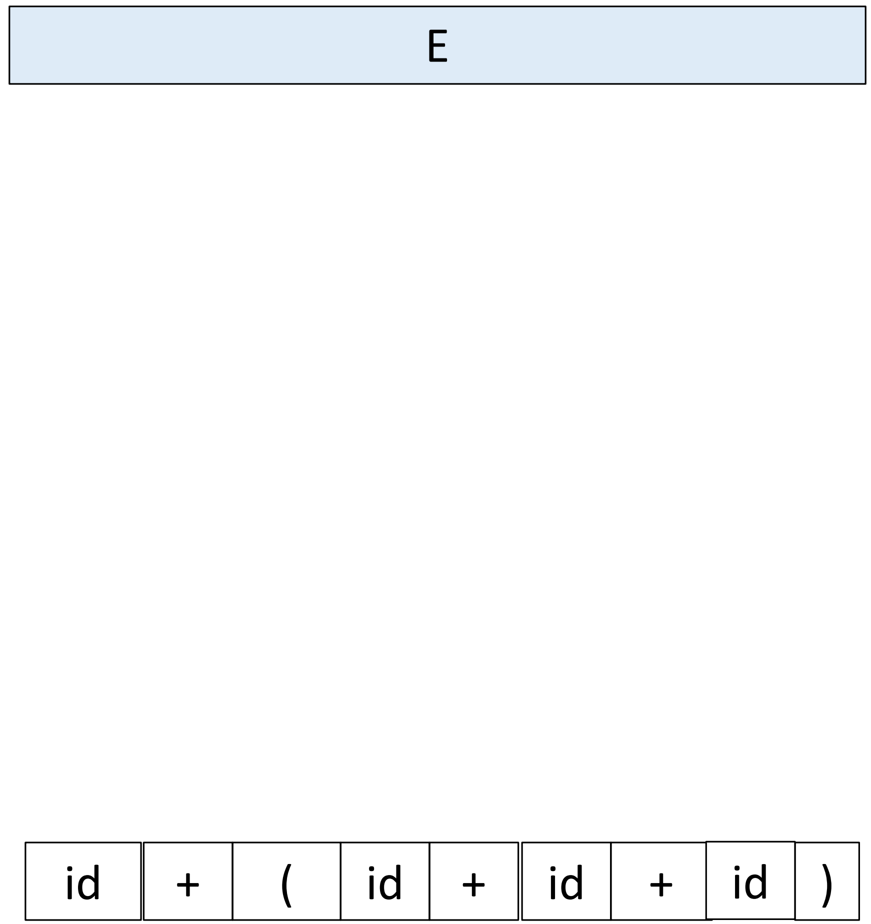
FINE!!!



Una terza vista sull'analisi bottom-up

$\Rightarrow$ E

FINE!!!



Analisi bottom-up: tre punti di vista (intuizioni)

- La prima intuizione (ricostruzione dell'analisi albero dal basso verso l'alto) motiva il modo in cui l'analisi dovrebbe funzionare
- La seconda intuizione (derivazione più a destra al contrario) descrive l'ordine in cui costruire l'albero di analisi
- La terza intuizione (individuare le produzioni da applicare) è la base per gli algoritmi di analisi bottom-up: quando si esegue una operazione di Reduce si sostituisce ad handle il (non terminale) lato sinistro della produzione

- Shift avanza nell'input alla ricerca di un handle
- Reduce sostituisce ad handle il non terminale a sinistra della produzione
- Un analizzatore left-to-right di tipo bottom-up ripetutamente cerca un handle e quindi lo riduce, fino a quando non completa la stringa di input e ottiene il solo simbolo iniziale

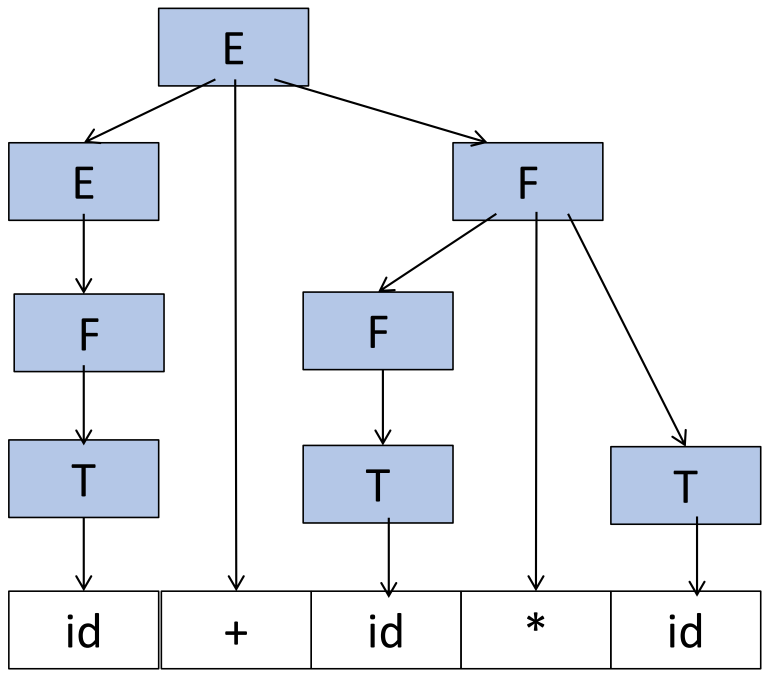
Handle

- Che algoritmo usiamo per trovarli?
- Una volta trovato un handle come sappiamo che è corretto?

Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

E → F
E → E + F
F → F * T
F → T
T → id
T → (E)



Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

Le risposte alle due domande non sono ovvie

$E \rightarrow F$

$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow id$

$T \rightarrow (E)$

id	+	id	*	id

Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

Nel seguito
- in rosso (a sinistra) la produzione oggetto di reduce
- In giallo (in basso) input esaminato (a seguito di operazioni di shift)

$E \rightarrow F$

$E \rightarrow E + F$

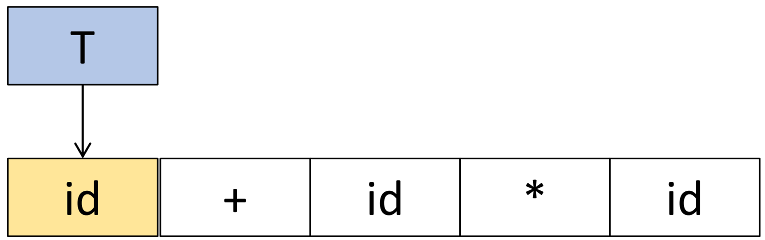
$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow id$

$T \rightarrow (E)$

Questo caso è facile: esiste una sola produzione con id nel lato destro

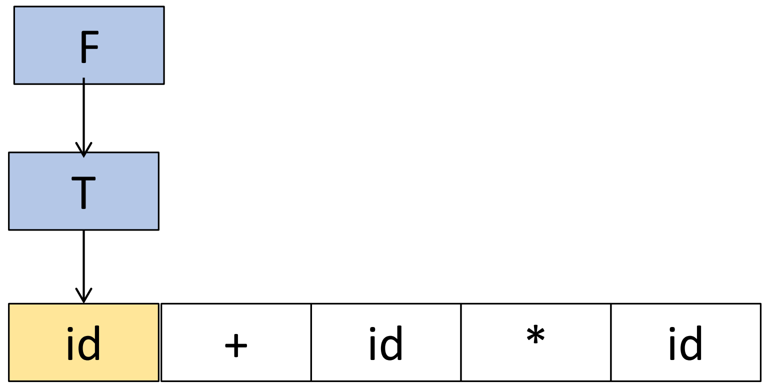


Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

E → F
E → E + F
F → F * T
F → T
T → id
T → (E)

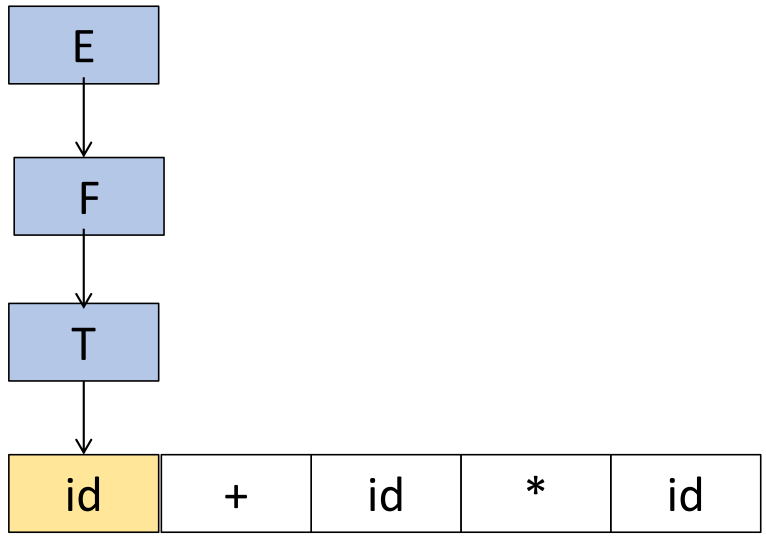
Anche questo caso è facile:
esiste una sola produzione con T nel lato destro all'inizio



Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

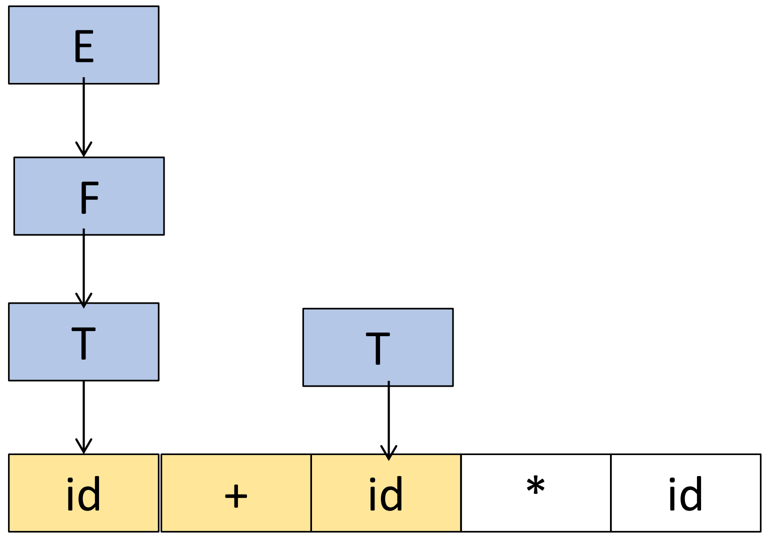
E → F
E → E + F
F → F * T
F → T
T → id
T → (E)



Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

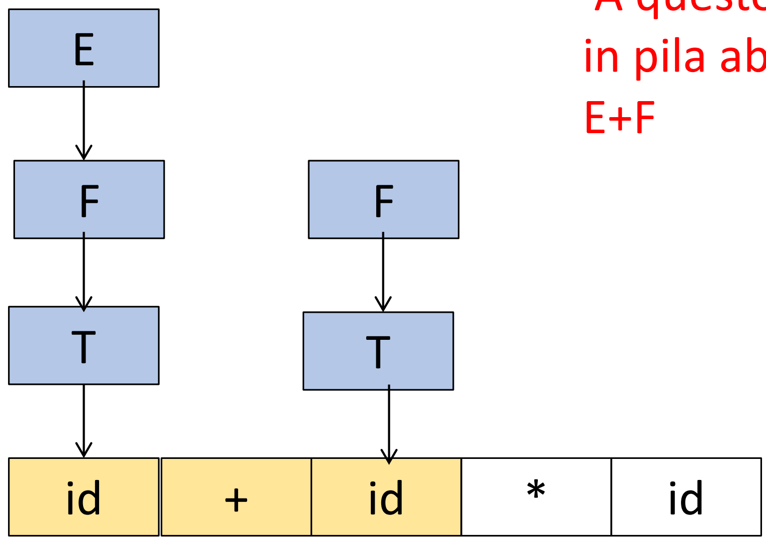
E → F
E → E + F
F → F * T
F → T
T → id
T → (E)



Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

$E \rightarrow F$
$E \rightarrow E + F$
$F \rightarrow F * T$
$F \rightarrow T$
$T \rightarrow id$
$T \rightarrow (E)$

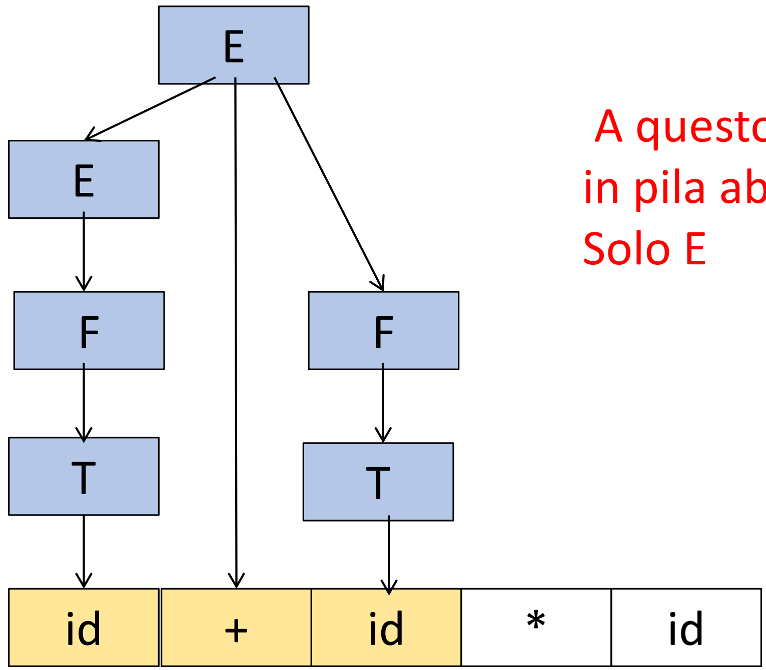


A questo punto in pila abbiamo E+F

Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

E → F
E → E + F
F → F * T
F → T
T → id
T → (E)

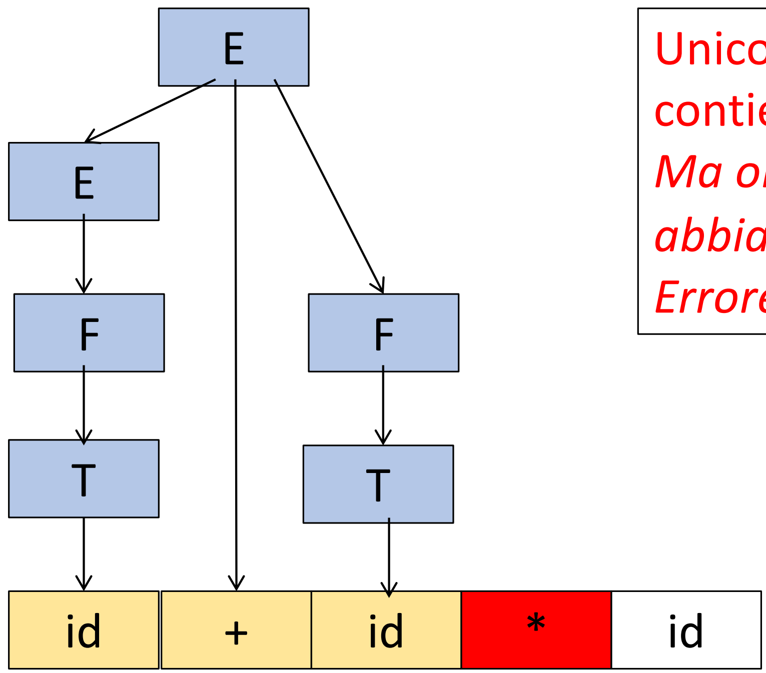


A questo punto in pila abbiamo Solo E

Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

E → F
E → E + F
F → F * T
F → T
T → id
T → (E)



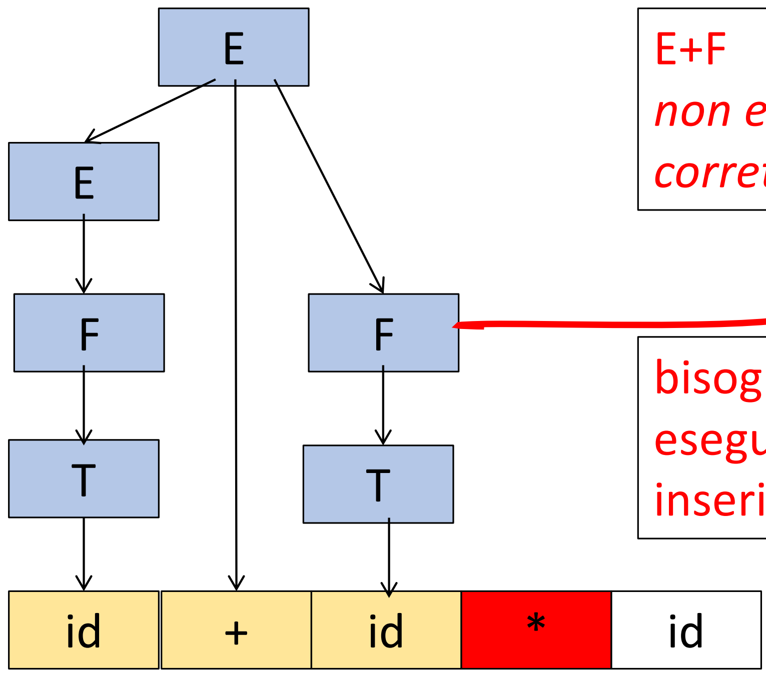
Unico handle che contiene * è F*T
Ma ora in pila abbiamo E
Errore!

Un esempio su handle

- Come trovare un handle?
- Una volta trovato come sappiamo se è corretto?

E → F
E → E + F
F → F * T
F → T
T → id
T → (E)

La riduzione più a sinistra non sempre fornisce l'handle corretto : conflitto shift -reduce



E+F
non era l'handle corretto!

bisognava eseguire shift e inserire * in pila

costruzione tavole Action e Goto

notazione punto ·

- si introduce un punto · per separare la parte destra di una produzione in due sottosequenze: a sinistra del punto · elementi già letti e impilati (shifted), a destra elementi ancora da analizzare
- es.: E → E·+T
  sono già stati impilati elementi derivati da E e occorre ora accettare in input il simbolo +
- ciascuna di queste produzioni con il punto si chiama *LR(0) item*, o semplicemente ***item***, e descrive lo stato del parser

grammatica aumentata

- Sia S il simbolo iniziale (o assioma); si aggiunge la produzione
  S'→S
- S' è il nuovo simbolo (che non compare in parti destre); l'obiettivo è avere l'assioma originale come parte destra di una produzione, così da poter effettuare una *reduce* conclusiva
- per ogni produzione A→α si considerano gli item ottenuti introducendo in tutte le posizioni possibili di α il puntino
- es. da E → E+T otteniamo 4 produzioni

  E → ·E+T, E → E·+T, E → E+·T, E → E+T ·

chiusura di un set di item

- solitamente non è possibile descrivere lo stato di un parser con un singolo item perché in presenza di ·X possono esistere vari elementi in First(X)
  - o elementi in Follow(X) qualora si annulli
- lo stato viene caratterizzato collezionando più item, attraverso una procedura detta di chiusura (**closure**)
- ad ogni collezione di item corrisponde uno stato

chiusura (closure) di un set di item

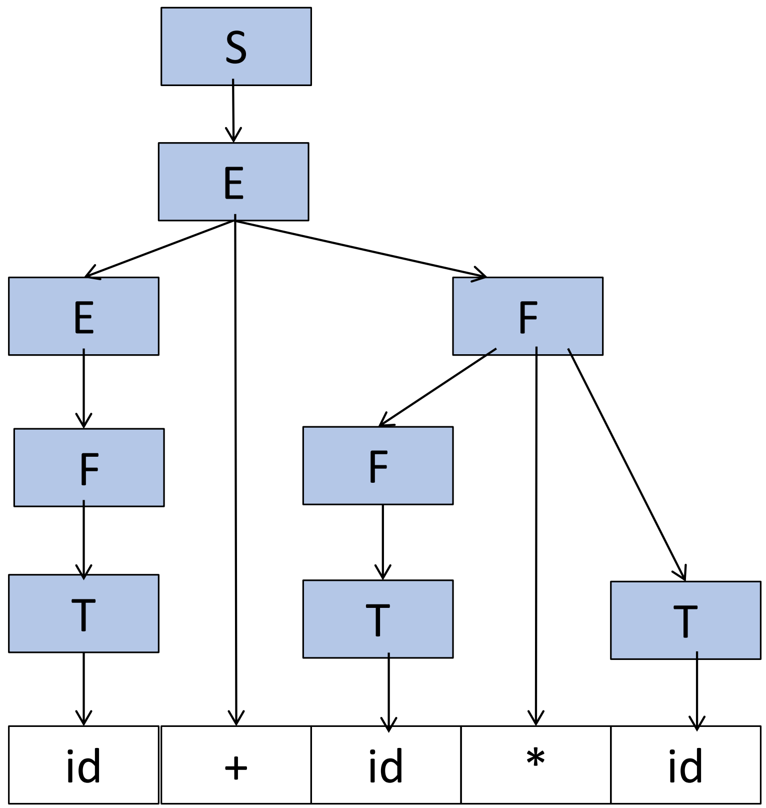
- se il set contiene un item $A \rightarrow \alpha \cdot B\beta$, $B \in V_N$, allora aggiungere al set $B \rightarrow \cdot\gamma$, per ciascuna produzione $B \rightarrow \gamma$, $\gamma \in V^*$
- continuare su tutte le produzioni nel set finché possibile, aggiungendo produzioni aventi come parte sinistra non-terminali che appaiono in parti destre preceduti dal punto
  - la parte dx di una produzione aggiunta inizierà con il punto

si definisce stato l'insieme risultante dall'operazione di chiusura di un set di item
(perfeziona e completa la def. informale già fornita prima)

Riprendiamo l'esempio precedente

S → E (S nuovo simbolo iniziale)
E → F
E → E + F
F → F * T
F → T
T → id
T → (E)

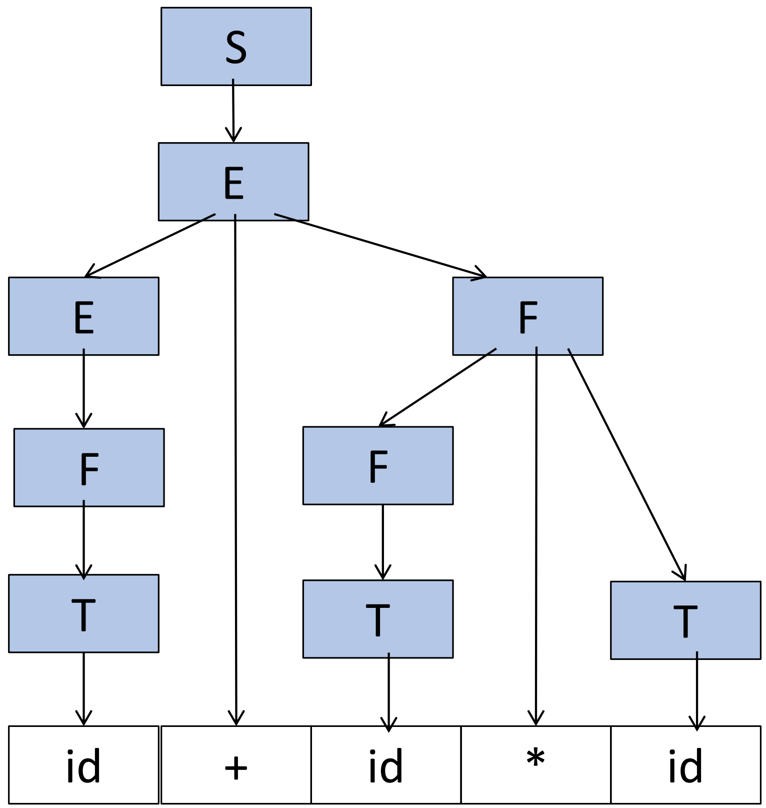
Prima abbiamo visto che non trovaiamo l'albero giusto e abbiamo errore quando arriviamo al terminale *



Riprendiamo l’esempio precedente

- Vediamo ora come la notazione punto risolve il problema
- La notazione punto permette di specificare nella pila a che punto siamo con una produzione

S → E (S nuovo simbolo iniziale)
E → F
E → E + F
F → F * T
F → T
T → id
T → (E)



Riprendiamo l’esempio precedente

Analizziamo solo fino a

S → E (S nuovo simbolo iniziale)
E → F
E → E + F
F → F * T
F → T
T → id
T → (E)

id	+	id	*	id

Riprendiamo l’esempio precedente

$S \rightarrow \cdot E$

S → E (S nuovo simbolo iniziale)

E → F

E → E + F

F → F * T

F → T

T → id

T → (E)

Per brevità assumiamo input sia in basso a destra (omettiamo produzioni inutili)

E	+	F	*	id

Riprendiamo l’esempio precedente

$S \rightarrow E$ (S nuovo simbolo iniziale)

$E \rightarrow F$

$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow id$

$T \rightarrow (E)$

Per brevità assumiamo input sia in basso a destra (omettiamo produzioni inutili)

$S \rightarrow \cdot E$
$E \rightarrow \cdot E + F$

id	+	id	*	id

Riprendiamo l'esempio precedente

S → E (S nuovo simbolo iniziale)

E → F

E → E + F

F → F * T

F → T

T → id

T → (E)

Per brevità assumiamo input sia in basso a destra (omettiamo produzioni inutili)

S → · E

E → · E + F

E → · E + F

E	+	F	*	id

Riprendiamo l’esempio precedente

$S \rightarrow E$ (S nuovo simbolo iniziale)

$E \rightarrow F$

$E \rightarrow E + F$

$F \rightarrow F * T$

$F \rightarrow T$

$T \rightarrow id$

$T \rightarrow (E)$

$S \rightarrow \cdot E$

$E \rightarrow \cdot E + F$

$E \rightarrow E \cdot + F$

Per brevità assumiamo input sia in basso a destra (omettiamo produzioni inutili)

id	+	id	*	id

Riprendiamo l’esempio precedente

S → E (S nuovo simbolo iniziale)
E → F
E → E + F
F → F * T
F → T
T → id
T → (E)

S → · E

E → · E + F

E → E + · F

eseguiamo shift

Per brevità assumiamo input sia in basso a destra (omettiamo produzioni inutili)

id	+	id	*	id

Riprendiamo l’esempio precedente

$S \rightarrow \cdot E$

$E \rightarrow \cdot E + F$

$E \rightarrow E + \cdot F$

$F \rightarrow F * \cdot T$

$T \rightarrow \cdot id$

E	+	F	*	id

Riprendiamo l’esempio precedente

S → E (S nuovo simbolo iniziale)
E → F
E → E + F
F → F * T
F → T
T → id
T → (E)

Per brevità assumiamo input sia in basso a destra (omettiamo produzioni inutili)

S → · E
E → · E + F
E → E + · F
F → F * · T
T → id

Abbiamo terminato analisi di T→id pop su pila

E	+	F	*	id

Riprendiamo l’esempio precedente

S → E (S nuovo simbolo iniziale)
E → F
E → E + F
F → F * T
F → T
T → id
T → (E)

Per brevità assumiamo input sia in basso a destra (omettiamo produzioni inutili)

S → · E
E → · E + F
E → E + · F
F → F * T ·

Abbiamo terminato analisi di F→ F * T

E	+	F	*	id

Riprendiamo l'esempio precedente

S → E (S nuovo simbolo iniziale)
E → F
E → E + F
F → F * T
F → T
T → id
T → (E)

Per brevità assumiamo input sia in basso a destra (omettiamo produzioni inutili)

S → · E
E → · E + F
E → E + F ·

Abbiamo terminato analisi di F→ F * T
....

E	+	F	*	id

Riprendiamo l’esempio precedente

- consideriamo la grammatica G' che genera il linguaggio

$L' = \{a^n b^m c^{n+m} \mid n+m > 0\}$

$S' \rightarrow S$

$S \rightarrow aSc \mid ac \mid T$

$T \rightarrow bTc \mid bc$

determiniamo i set of item

$S' \rightarrow S$
$S \rightarrow aSc \mid ac \mid T$
$T \rightarrow bc \mid bTc$

- inizio: assioma $S' \rightarrow \cdot S$
- chiusura*:*
  - $S' \rightarrow \cdot S$
  - (+) $S \rightarrow \cdot aSc \mid \cdot T \mid \cdot ac \mid$
- questa collezione di produzioni definisce uno stato (s_0)
- il simbolo (+) indica che la riga di produzioni è aggiunta durante la chiusura

per definire gli altri item sets si procede ricorsivamente

- per ogni produzione $A \rightarrow \alpha \cdot B\beta$, $B \in V_N$, si crea un nuovo stato (se non già creato) contenente la produzione $A \rightarrow \alpha B \cdot \beta$, e se ne fa la chiusura
- il simbolo "scavalcato" dal punto determina la transizione fra stati

$S' \rightarrow S$
$S \rightarrow aSc \mid ac \mid T$
$T \rightarrow bc \mid bTc$

determinazione altri stati

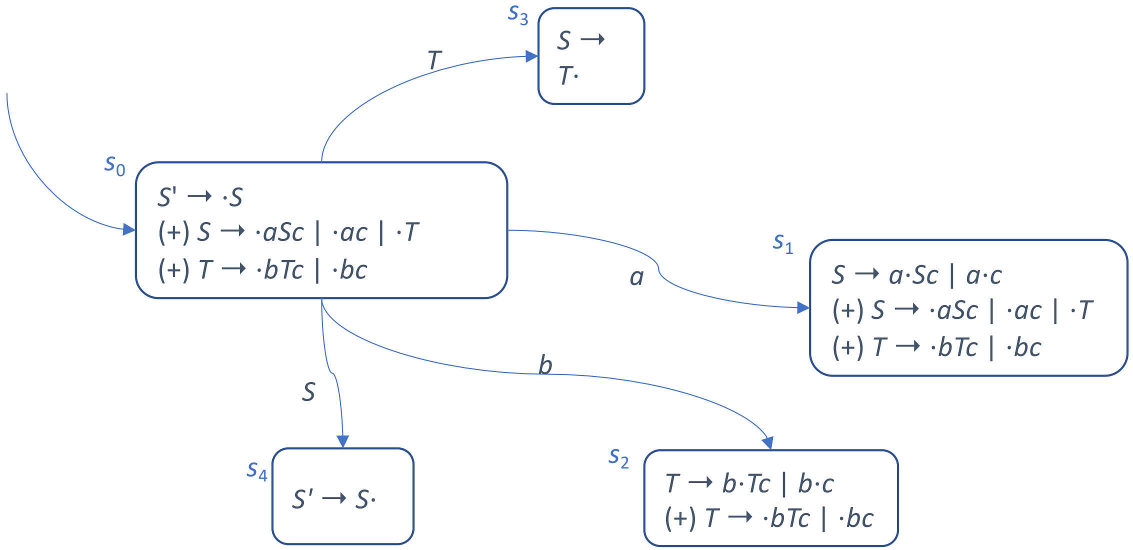
stato s_0

$S' \rightarrow \cdot S$
(+) $S \rightarrow \cdot aSc \mid \cdot T \mid \cdot ac$
(+) $T \rightarrow \cdot bTc \mid \cdot bc$

- da s_0 a s_1 (simbolo a)
  $S \rightarrow a \cdot Sc \mid a \cdot c$
  (+) $S \rightarrow \cdot aSc \mid \cdot ac \mid \cdot T$
  (+) $T \rightarrow \cdot bTc \mid \cdot bc$
- da s_0 a s_2 (simbolo b)
  $T \rightarrow b \cdot Tc \mid b \cdot c$
  (+) $T \rightarrow \cdot bTc \mid \cdot bc$
- da s_0 a s_3 (simbolo T)
  $S' \rightarrow T \cdot$
- da s_0 a s_4 (simbolo S)
  $S' \rightarrow S \cdot$

goto-graph (transition diagram)

$S' \to S$
$S \to aSc \mid ac \mid T$
$T \to bc \mid bTc$



$S' \to S$
$S \to aSc \mid ac \mid T$
$T \to bc \mid bTc$

determinazione altri stati

stato s_1 (simbolo a)

$S \to a{\cdot}Sc \mid T{\cdot} \mid a{\cdot}c$
(+) $S \to {\cdot}aSc \mid {\cdot}T \mid {\cdot}ac$

(+) $T \to {\cdot}bTc \mid {\cdot}bc$

da s_1 a s_1 (simbolo a) – chiusura fornisce lo stesso insieme

da s_1 a s_2 (simbolo b)
$T \to b{\cdot}Tc \mid b{\cdot}c$
(+) $T \to {\cdot}bTc \mid {\cdot}bc$

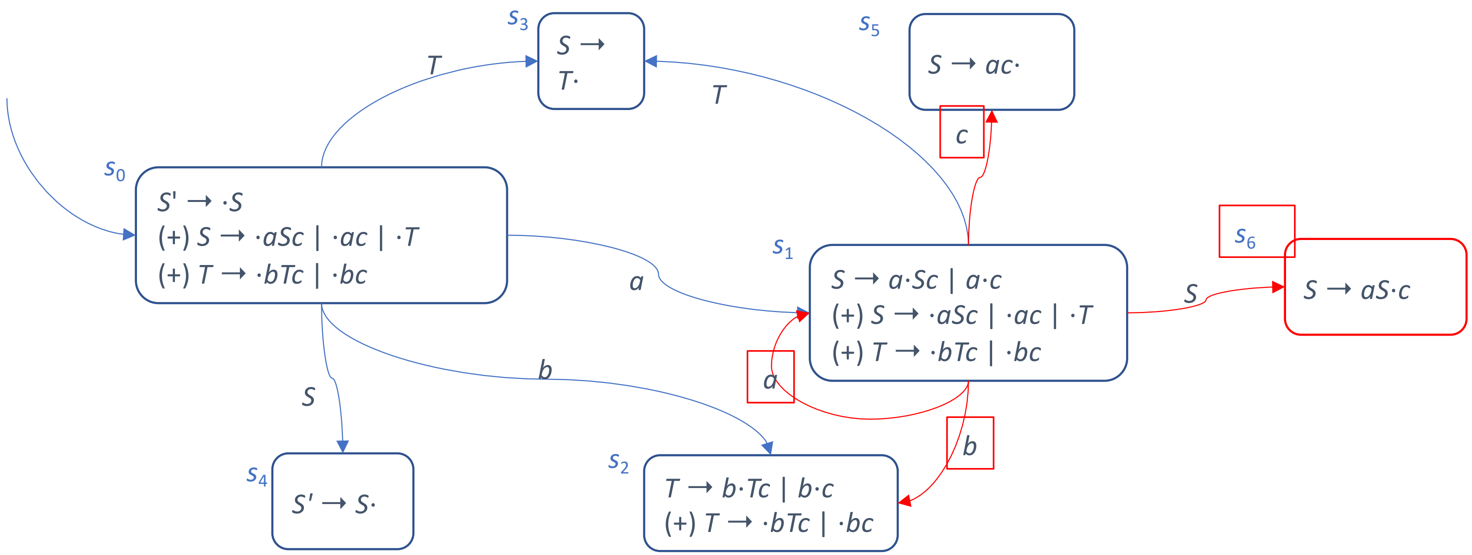
da s_1 a s_3 (simbolo T)
$S \to aT{\cdot}c$

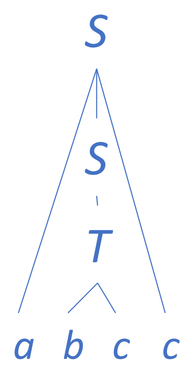
da s_1 a s_5 (simbolo c)
$S \to a\ c{\cdot}$

da s_1 a s_6 (simbolo S)
$S \to T{\cdot}$

goto-graph (transition diagram)

$S' \to S$
$S \to aSc \mid ac \mid T$
$T \to bc \mid bTc$





determinazione altri stati

$S' \rightarrow S$
$S \rightarrow aSc \mid ac \mid T$
$T \rightarrow bc \mid bTc$

- stato s_2 (simbolo b)

$T \rightarrow b \cdot Tc \mid b \cdot c$

(+) $T \rightarrow \cdot bTc \mid \cdot bc$

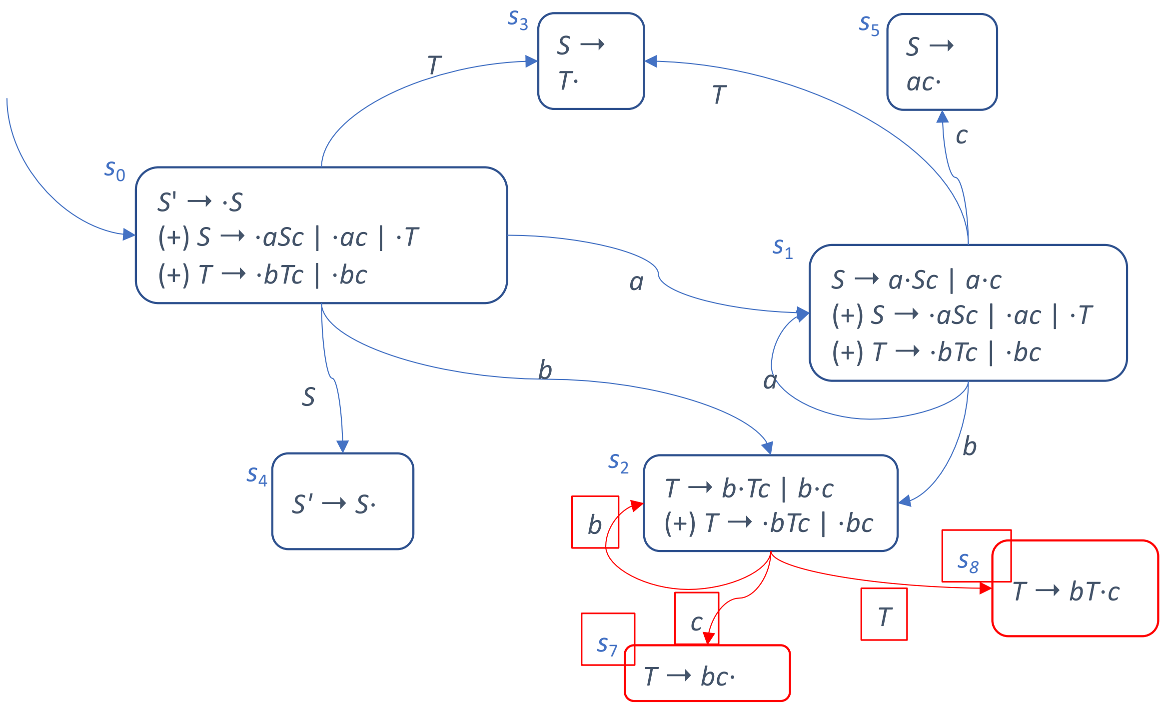
da s_2 a s_2 (simbolo b)

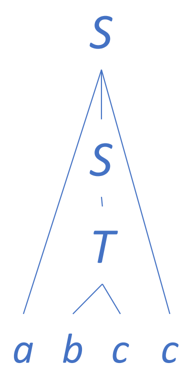
da s_2 a s_7 (simbolo c)
$S \rightarrow bc \cdot$

da s_2 a s_8 (simbolo T)
$S \rightarrow bT \cdot c$

goto-graph (transition diagram)

$S' \rightarrow S$
$S \rightarrow aSc \mid ac \mid T$
$T \rightarrow bc \mid bTc$





determinazione altri stati

$S' \rightarrow S$
$S \rightarrow aSc \mid ac \mid T$
$T \rightarrow bc \mid bTc$

stato s_6 (simbolo S)
$S \rightarrow a\, S{\cdot}c$

da s_6 a s_9 (simbolo c)
$S \rightarrow a\, S\, c{\cdot}$

stato s_8 (simbolo T)
$T \rightarrow b\, T{\cdot}c$

da s_8 a s_{10} (simbolo c)
$T \rightarrow bT\, c{\cdot}$

determinazione altri stati

$S' \to S$
$S \to aSc \mid ac \mid T$
$T \to bc \mid bTc$

stato s_{12} (simbolo T)
$T \to bT \cdot c$

stato s_{13} (simbolo b)
$T \to b \cdot Tc \mid b \cdot c$
(+) $T \to \cdot bTc \mid \cdot bc$

da s_{12} a s_{14} (simbolo c)
$T \to bTc \cdot$

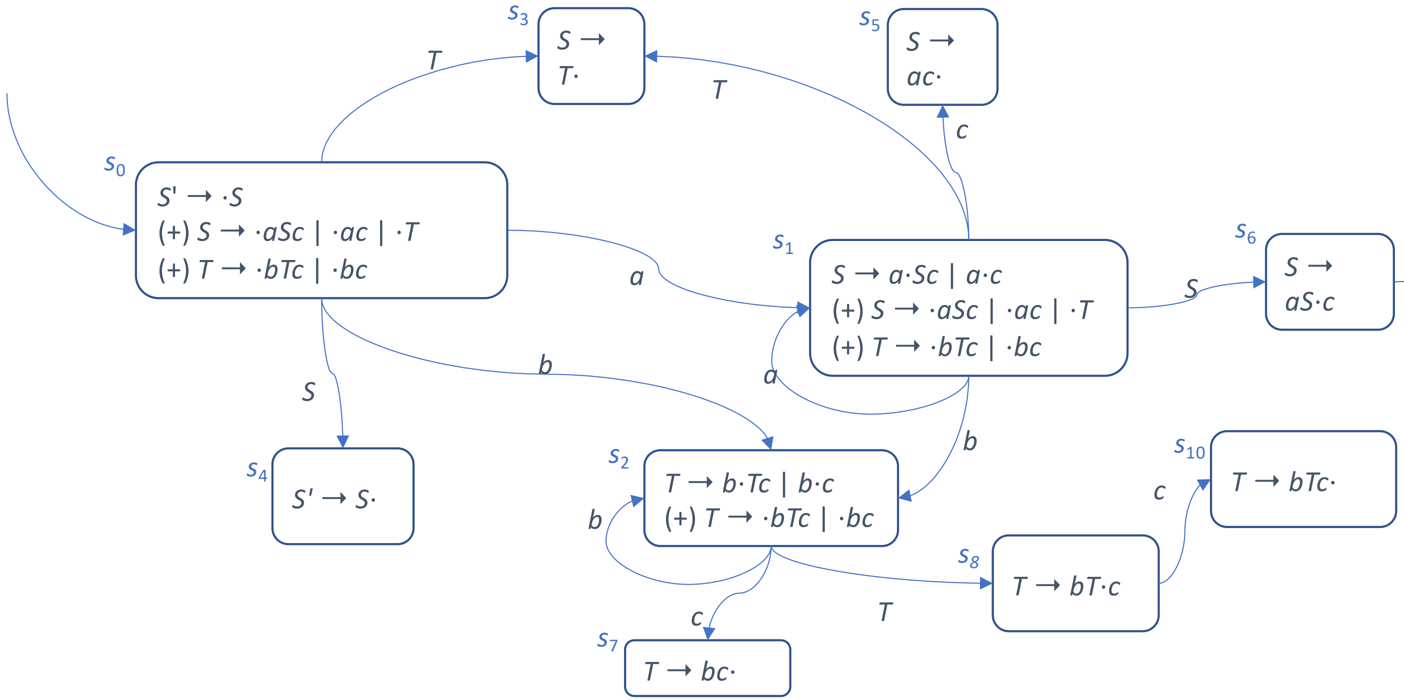
da s_{13} a s_{12} (simbolo T)

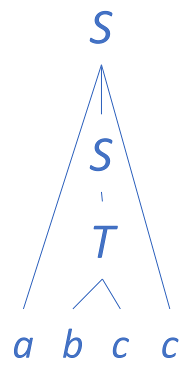
da s_{13} a s_{15} (simbolo c)
$T \to bc \cdot$

da s_{13} a s_{13} (simbolo b)

goto-graph (transition diagram)

$S' \to S$
$S \to aSc \mid ac \mid T$
$T \to bc \mid bTc$





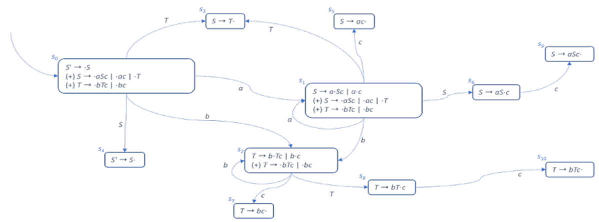
Tavole Action e Goto

$$S' \to S$$
$$S \to aSc \mid ac \mid T$$
$$T \to bc \mid bTc$$

- Un arco fra due stati s_a e s_b etichettato con simbolo terminale x: equivale a passare da s_a a s_b quando input è x (shift): si inserisce s_b in pila
- Quando si giunge in uno stato con . alla fine bisogna eseguire reduce: si tolgono tanti stati dalla pila quanti sono i simboli a destra della produzione e si inserisce un nuovo stato in pila (tavola Goto)
- Goto[s,U]: dice quale stato inserire in pila quando lo stato affiorante è s e il simbolo a sinistra della produzione oggetto di reduce è U (in questo modo tengo conto della storia pregressa)

stato	*a*	*b*	*c*	*$*	*S*	*T*
0	*shift* 1	*shift* 2			*goto* 4	*goto* 3
1	*shift* 1	*shift* 2	*shift* 5		*goto* 6	*goto* 3
2		*shift* 2	*shift* 7			*goto* 8
3	*reduce* $S \to T$					
4				*accept*		
5	*reduce* $S \to ac$					
6			*shift* 9			
7	*reduce* $T \to bc$					
8			*shift* 10			
9	*reduce* $S \to aSc$					
10	*reduce* $T \to bTc$					
	Tavola Action				Tavola Goto	

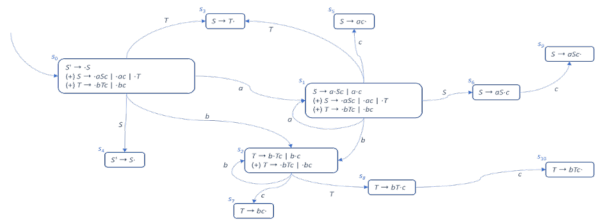
Tavole Action e Goto



- G'': $S' \rightarrow S$
  $S \rightarrow aSc \mid ac \mid T$
  $T \rightarrow bc \mid bTc$
- parse di *abcc* \$
- (*shift* = s, *reduce* = r, *goto* = g)
- s(1), s(2), s(7), r($T\rightarrow bc$), g(3), r($S\rightarrow T$), g(6), s(9), r($S\rightarrow aSc$), g(4)
- in input rimane *\$* (fine input), per cui: ***accept***

stato	*a*	*b*	*c*	*\$*	*S*	*T*
0	*shift* 1	*shift* 2			*goto* 4	*goto* 3
1	*shift* 1	*shift* 2	*shift* 5		*goto* 6	*goto* 3
2		*shift* 2	*shift* 7			*goto* 8
3	*reduce* $S \rightarrow T$					
4				*accept*		
5	*reduce* $S \rightarrow ac$					
6			*shift* 9			
7	*reduce* $T \rightarrow bc$					
8			*shift* 10			
9	*reduce* $S \rightarrow aSc$					
10	*reduce* $T \rightarrow bTc$					
	Tavola Action				**Tavola Goto**	

Tavole Action e Goto



- G'': $S' \rightarrow S$
  $S \rightarrow aSc \mid ac \mid T$
  $T \rightarrow bc \mid bTc$
- Input *abcc$*

Pila input (da esaminare)

s0 *abcc$*

stato	*a*	*b*	*c*	*$*	*S*	*T*
0	*shift* 1	*shift* 2			*goto* 4	*goto* 3
1	*shift* 1	*shift* 2	*shift* 5		*goto* 6	*goto* 3
2		*shift* 2	*shift* 7			*goto* 8
3	*reduce* $S \rightarrow T$					
4				*accept*		
5	*reduce* $S \rightarrow ac$					
6			*shift* 9			
7	*reduce* $T \rightarrow bc$					
8			*shift* 10			
9	*reduce* $S \rightarrow aSc$					
10	*reduce* $T \rightarrow bTc$					
	Tavola Action				**Tavola Goto**	

goto-graph (diagramma transizioni):
altro esempio

Riconsideriamo la grammatica

$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow id$

$T \rightarrow (E)$

s_0

$\boxed{S \rightarrow .\, E}$

Inizio: metti produzione da simbolo iniziale

goto-graph (diagramma transizioni)

$S \rightarrow E$

$E \rightarrow T;$

$E \rightarrow T + E$

$T \rightarrow id$

$T \rightarrow (E)$

s_0

$S \rightarrow .\ E$

$E \rightarrow .\ T;$

$E \rightarrow .\ T + E$

Chiudi su E

goto-graph (diagramma transizioni)

$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow id$
$T \rightarrow (E)$

s_0

$S \rightarrow . E$
$E \rightarrow . T;$
$E \rightarrow . T + E$
$T \rightarrow . id$
$T \rightarrow . (E)$

Completa la chiusura
includendo T

goto-graph (diagramma transizioni)

$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow id$
$T \rightarrow (E)$

s_0

$S \rightarrow .\, E$
$E \rightarrow .\, T;$
$E \rightarrow .\, T + E$
$T \rightarrow .\, id$
$T \rightarrow .\, (E)$

Scelgo la prima produzione
(reduce)
Sposto il punto

E

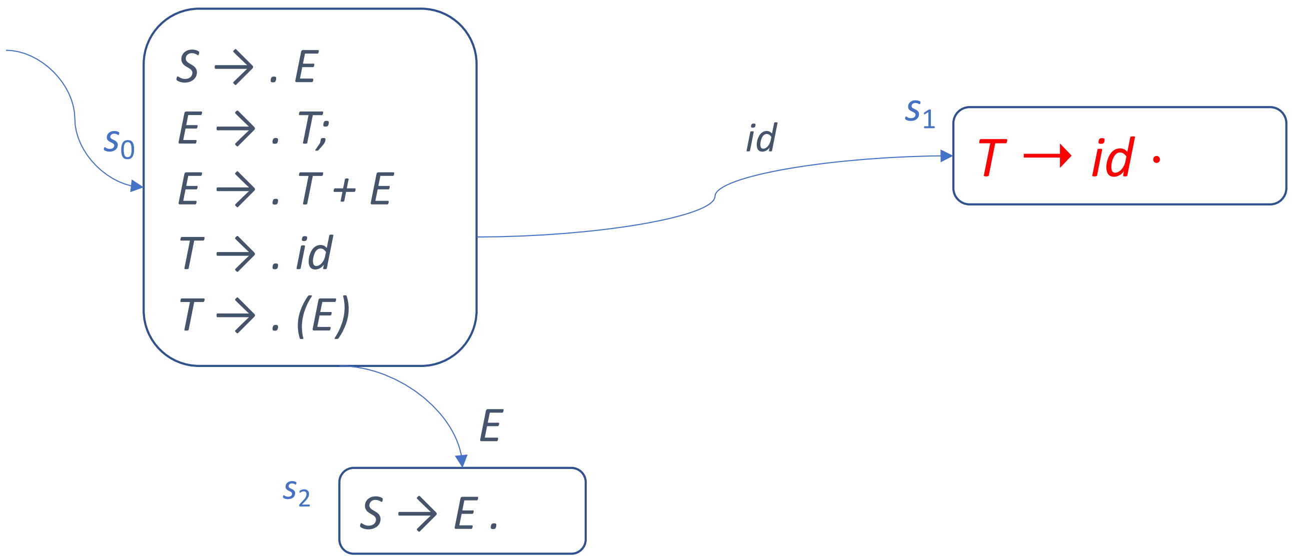
s_2

$S \rightarrow E\, .$

La produzione è completata
(abbiamo il . alla fine)

goto-graph (diagramma transizioni)

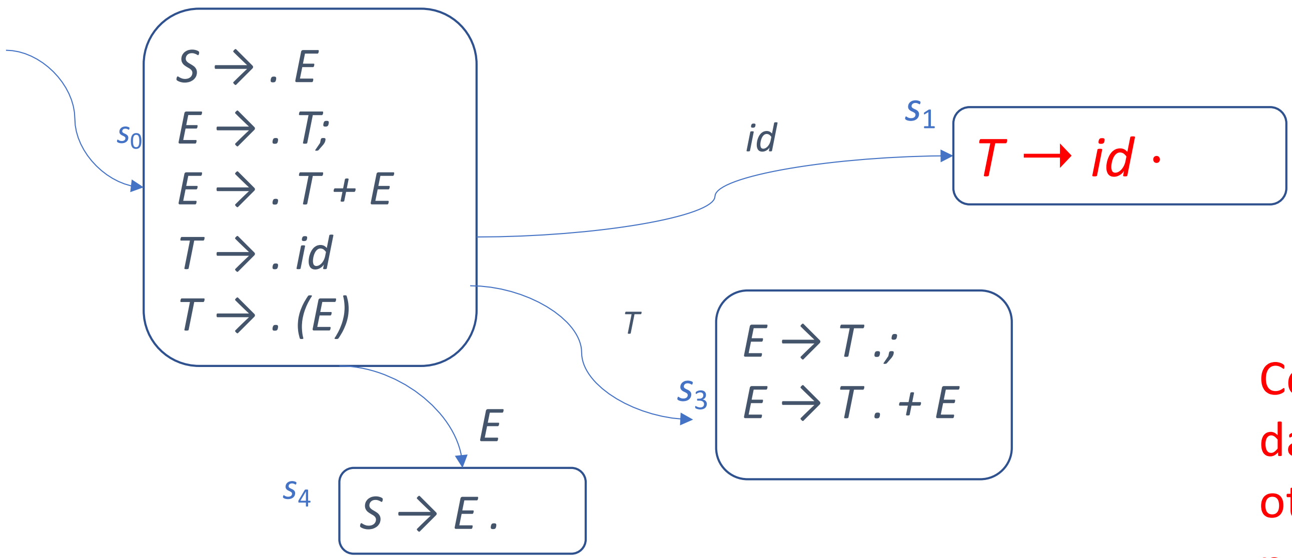
S → E
E → T;
E → T + E
T → id
T → (E)



Avanzo in input (shift)
Sposto il punto

goto-graph (diagramma transizioni)

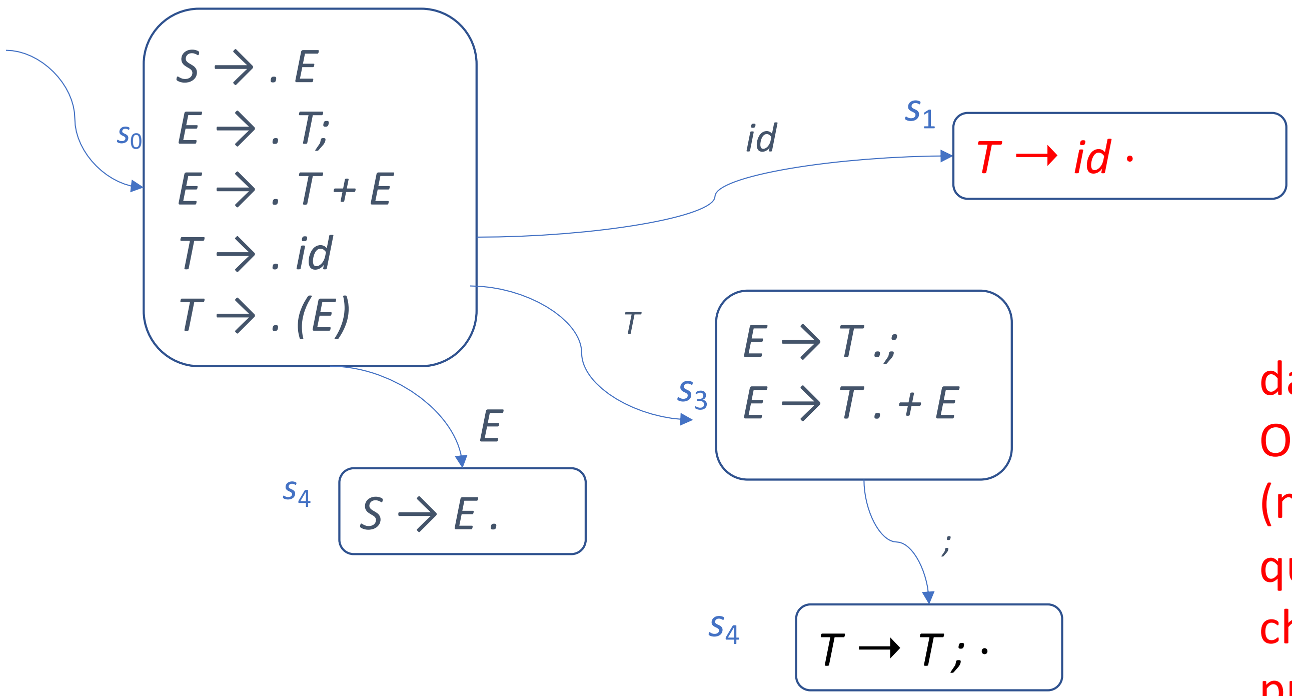
$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow id$
$T \rightarrow (E)$



Continuo in modo analogo da s0 con E→T. e E→ T. + E ottengo (non devo chiudere perché dopo il punto simboli terminali)

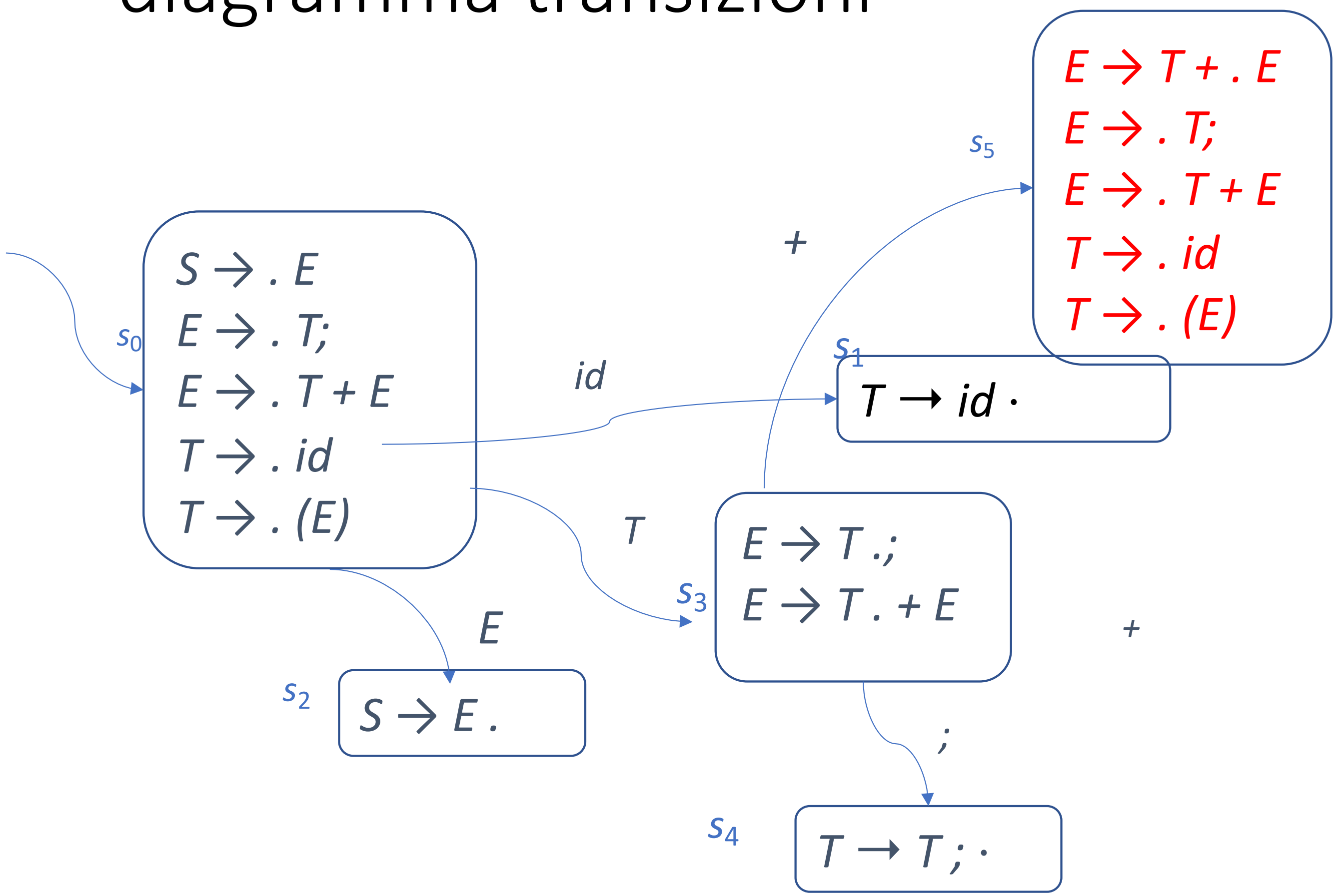
goto-graph (diagramma transizioni)

$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow id$
$T \rightarrow (E)$



da s0 con $E \rightarrow T . ;$
Ottengo $T \rightarrow T; .$
(nota il punto alla fine:
questo è uno stato che indica
che siamo alla fine dell'analisi
produzione)

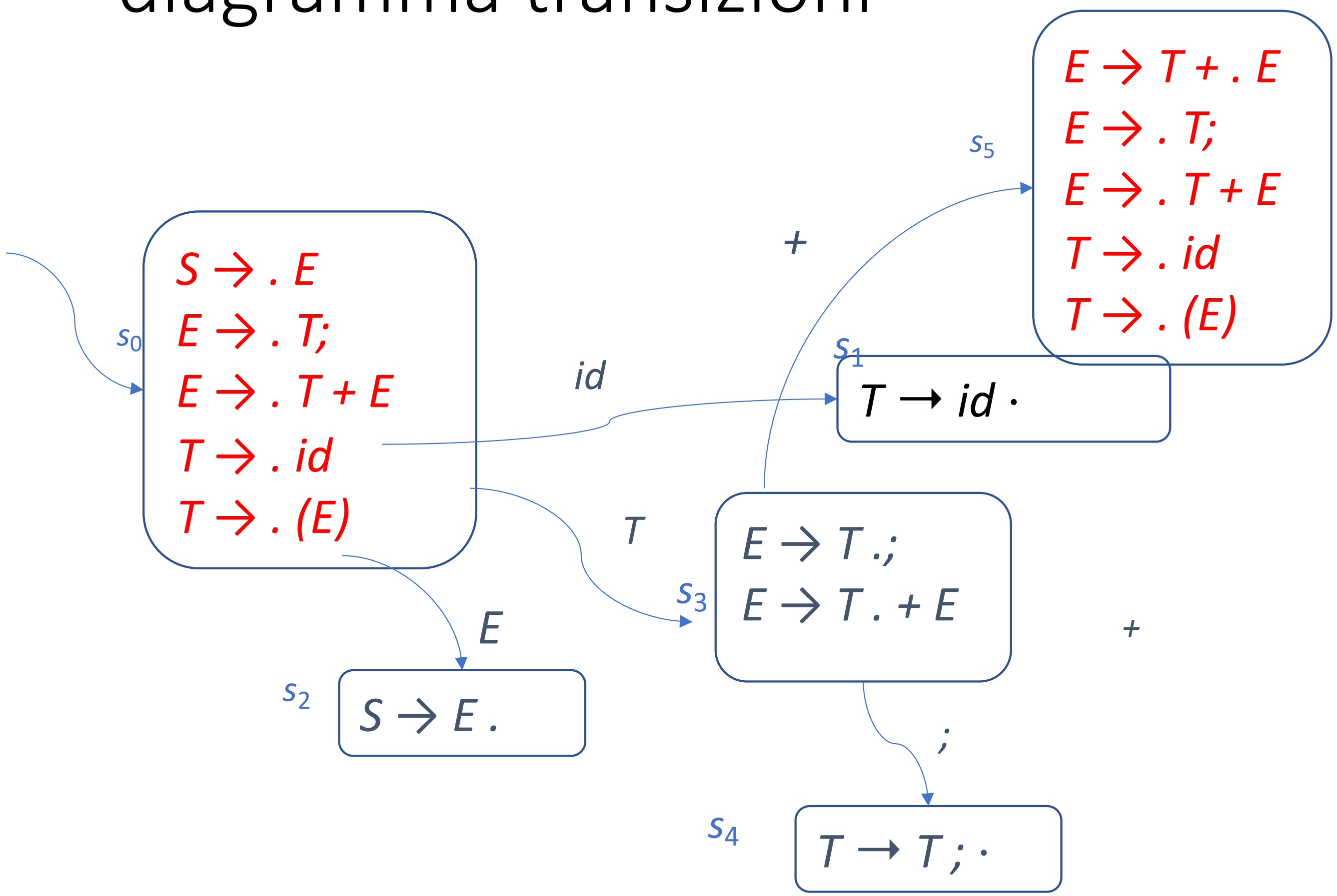
diagramma transizioni



S → E
E → T;
E → T + E
T → id
T → (E)

Continuo in modo analogo
da E→T. +E ottengo
(ricorda faccio chiusura)

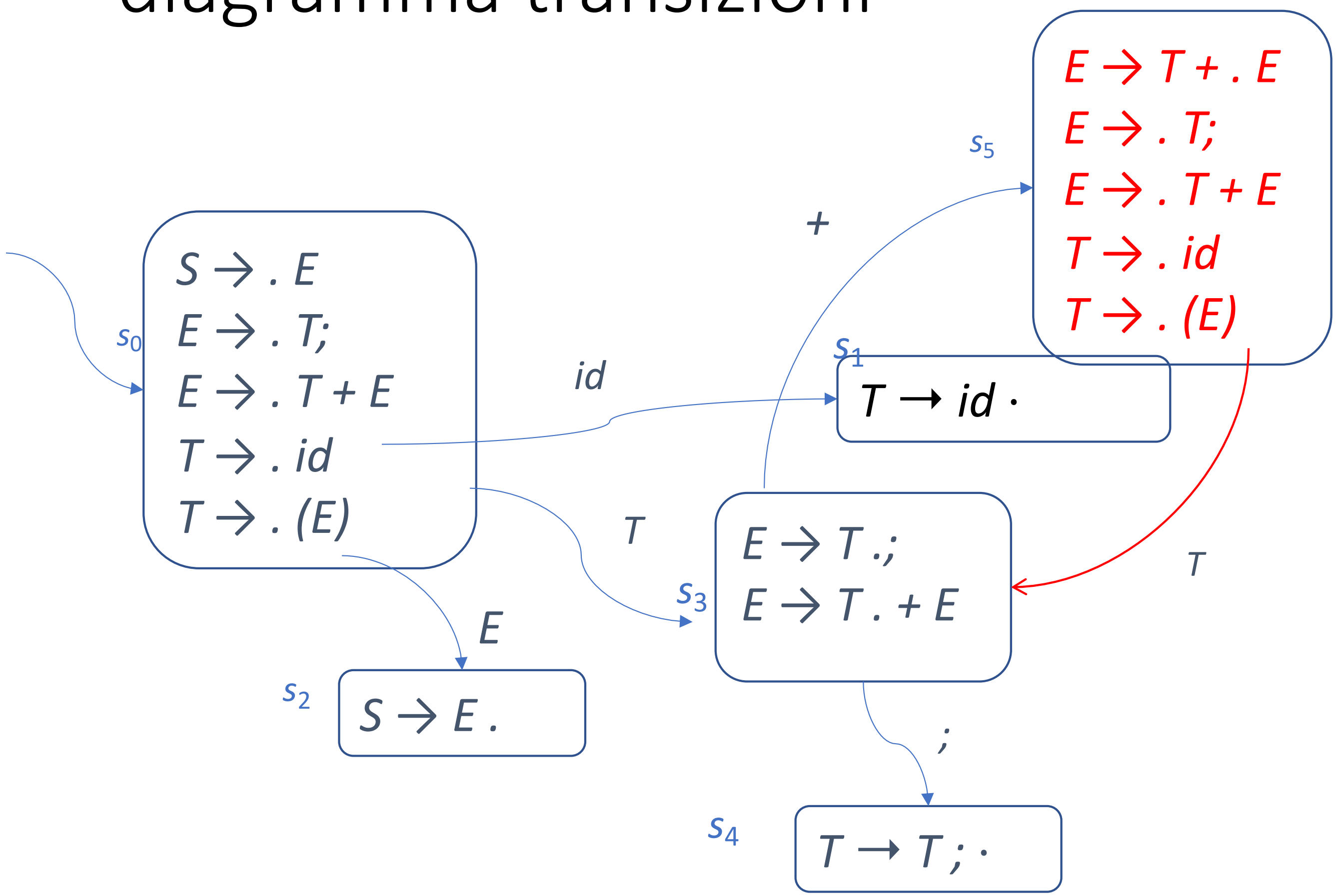
diagramma transizioni



$S \to E$
$E \to T;$
$E \to T + E$
$T \to id$
$T \to (E)$

NOTA: gli stati s0 e s5 sono quasi identici
s5 non ha $S \to . E$
s1 non ha $E \to T + . E$
Usiamo la notazione punto per cui $E \to T + . E$
è diversa da $E \to T + . E$ *(che a sua volta è uno degli stati di s1)*

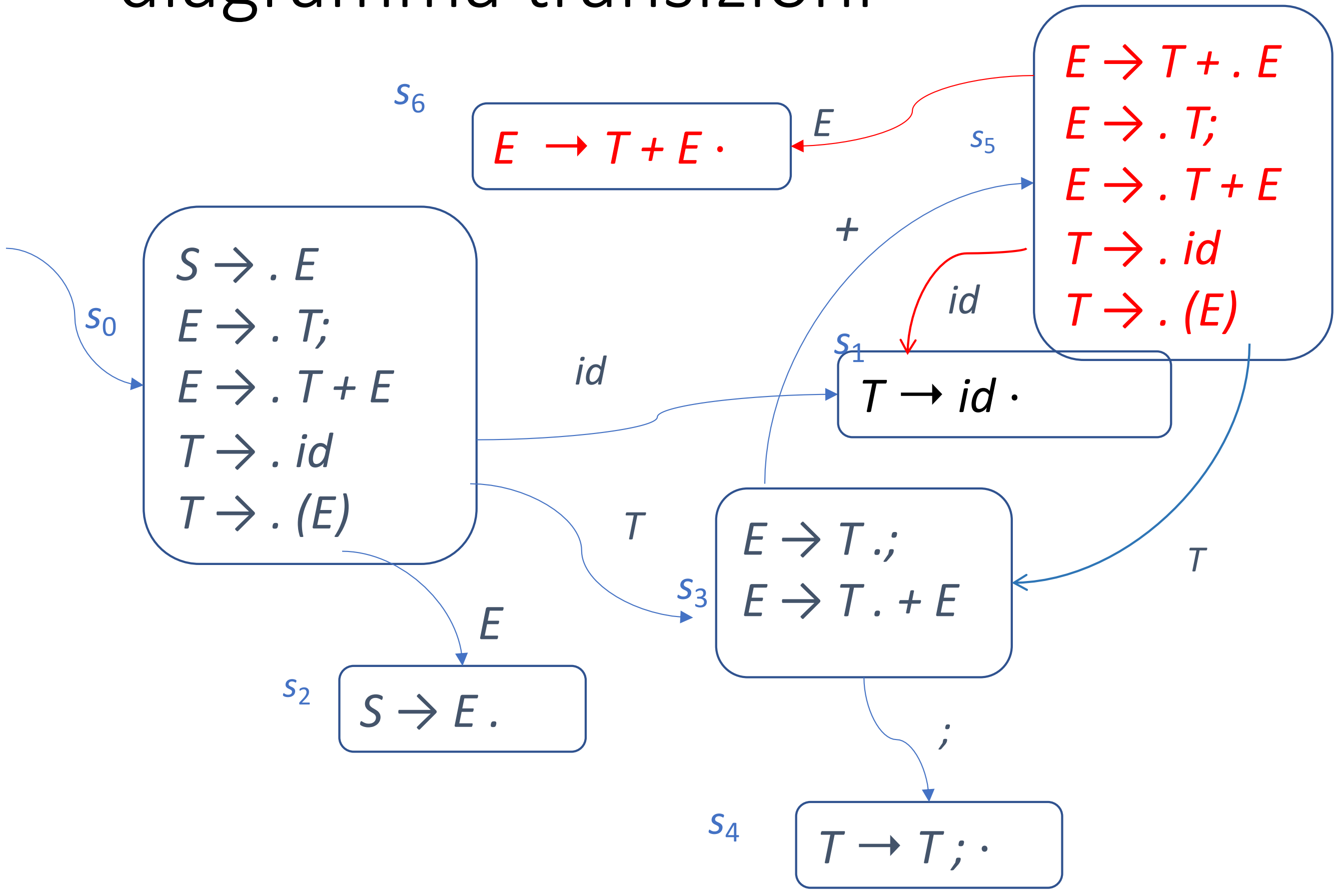
diagramma transizioni



$S \rightarrow E$
$E \rightarrow T;$
$E \rightarrow T + E$
$T \rightarrow id$
$T \rightarrow (E)$

da s5 e T con
$E\rightarrow . T ;$ e $E\rightarrow . T + E$
ottengo
$E\rightarrow T . ;$ e $E\rightarrow T . + E$
Stato già trovato!

diagramma transizioni

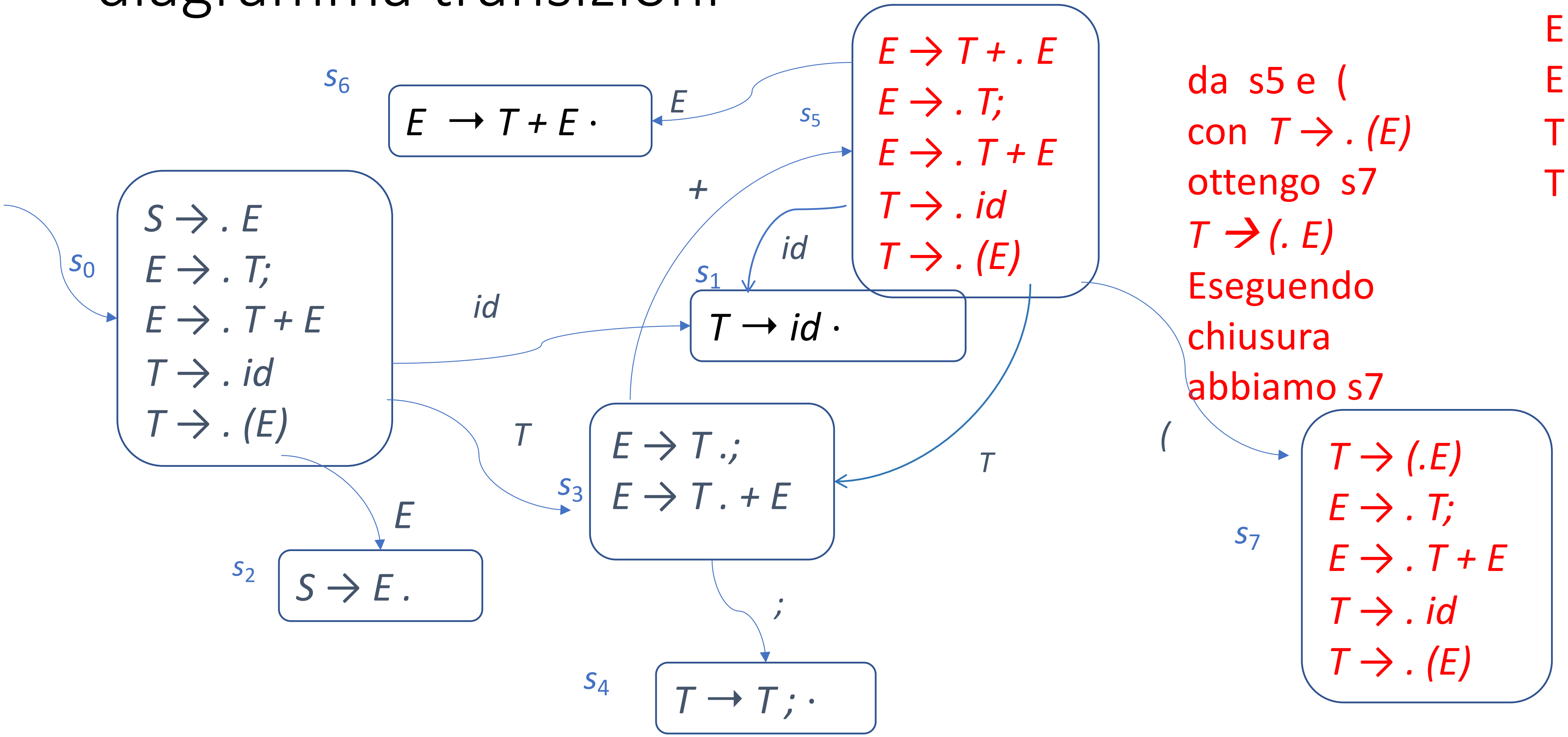


S → E
E → T;
E → T + E
T → id
T → (E)

da s5 e E
con E→ T + . E
ottengo s6
E→ T + E .

da s5 e id
Con *T* → . *Id*
ottengo s1 T→id .
stato già trovato

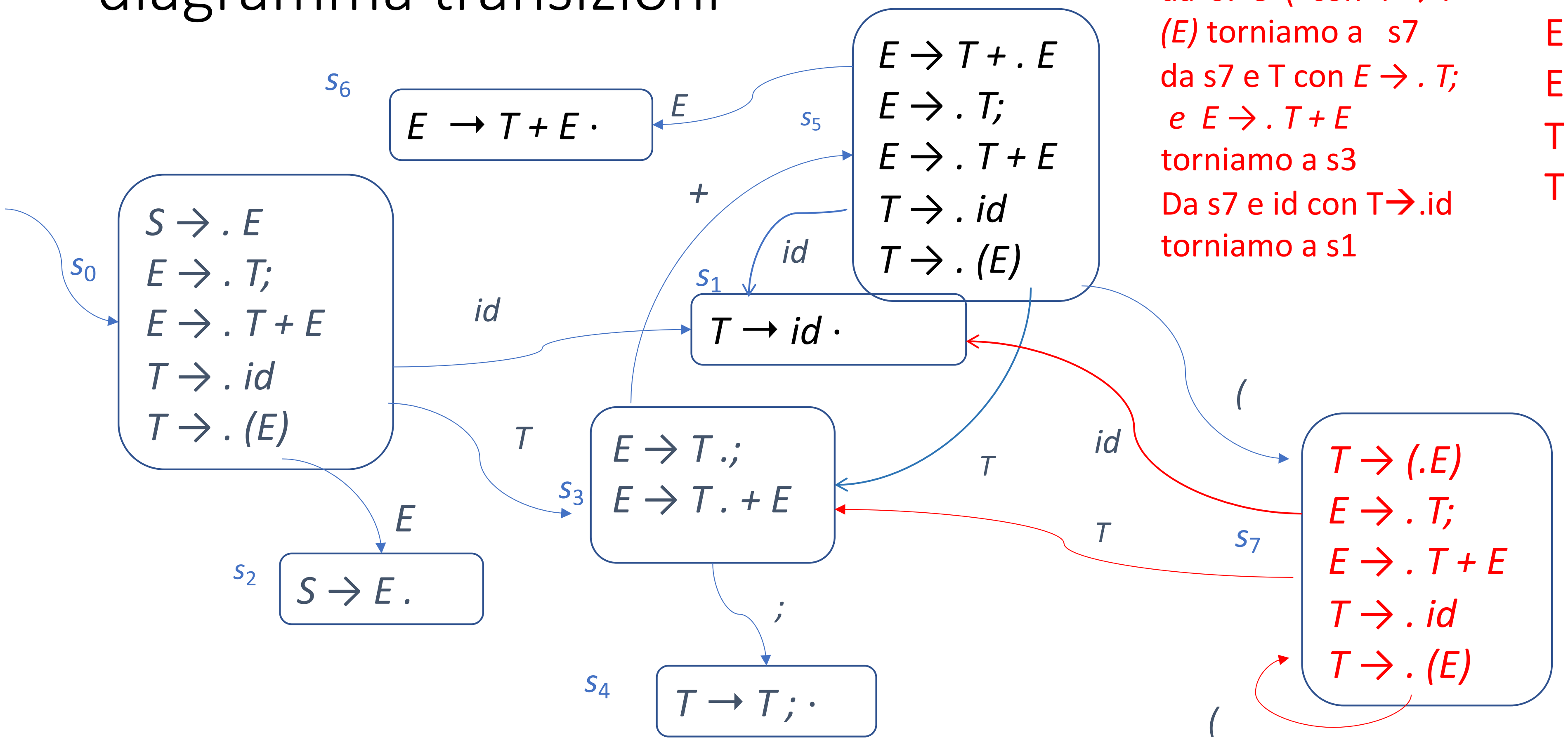
diagramma transizioni



da s5 e (
con $T \to . (E)$
ottengo s7
$T \to (. E)$
Eseguendo
chiusura
abbiamo s7

$S \to E$
$E \to T;$
$E \to T + E$
$T \to id$
$T \to (E)$

diagramma transizioni



da s7 e (con $T \rightarrow .(E)$ torniamo a s7

da s7 e T con $E \rightarrow . T;$ e $E \rightarrow . T + E$ torniamo a s3

Da s7 e id con T→.id torniamo a s1

S → E
E → T;
E → T + E
T → id
T → (E)

diagramma transizioni

Proseguendo da s7
otteniamo s8 e s9

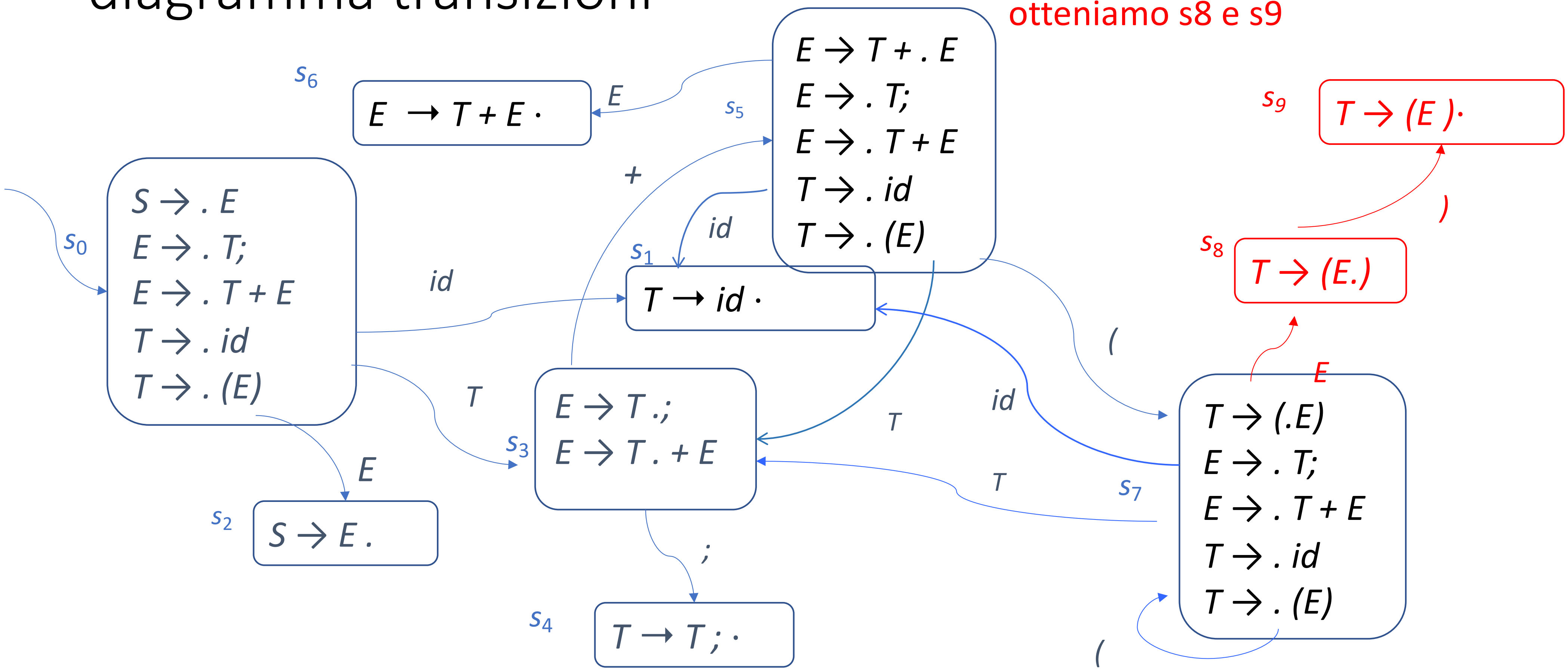


diagramma transizioni: abbiamo finito?

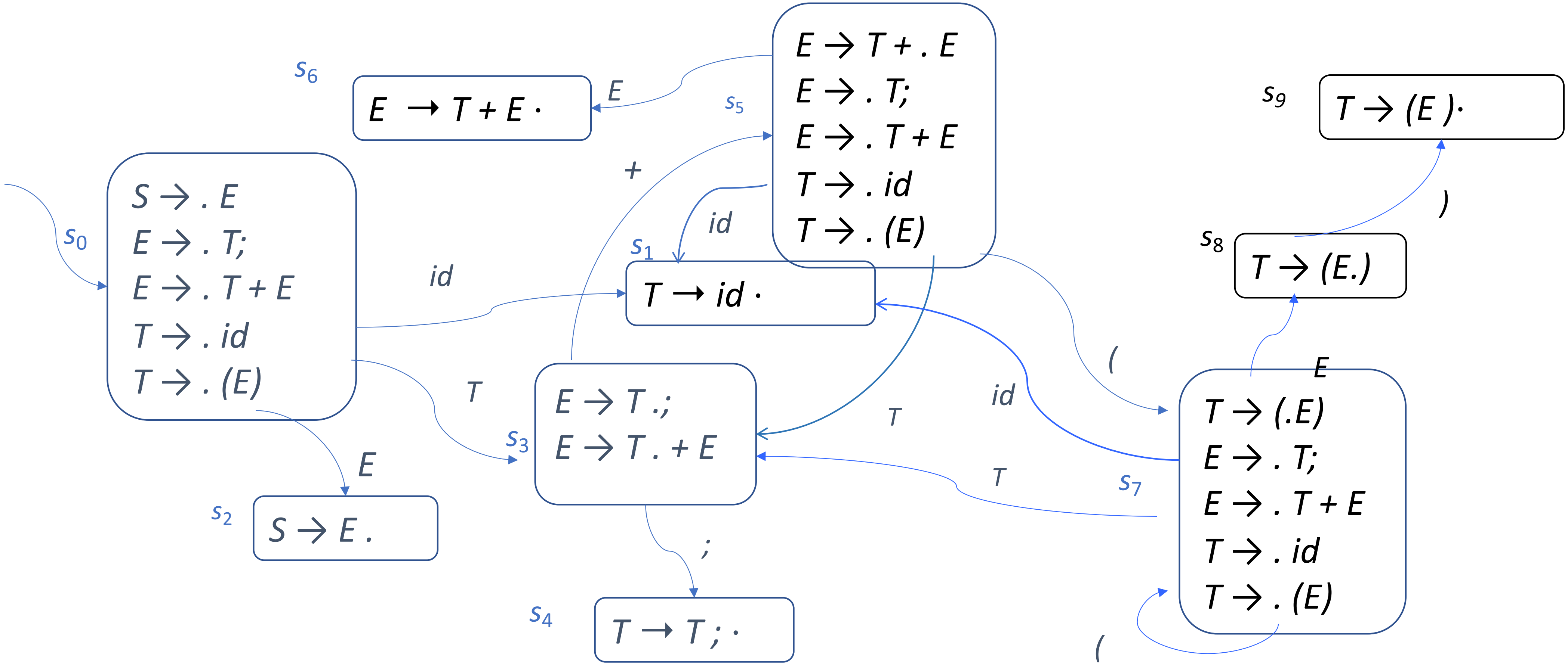


diagramma transizioni: abbiamo finito?

Abbiamo finito la costruzione quando abbiamo considerato per ogni stato tutte le possibili transizioni in corrispondenza a ciascuna produzione associata allo stato tenendo conto della notazione punto ed escludendo quelle che nella parte destra finiscono con il punto)

Questo è vero per s0? NO MANCA (!!

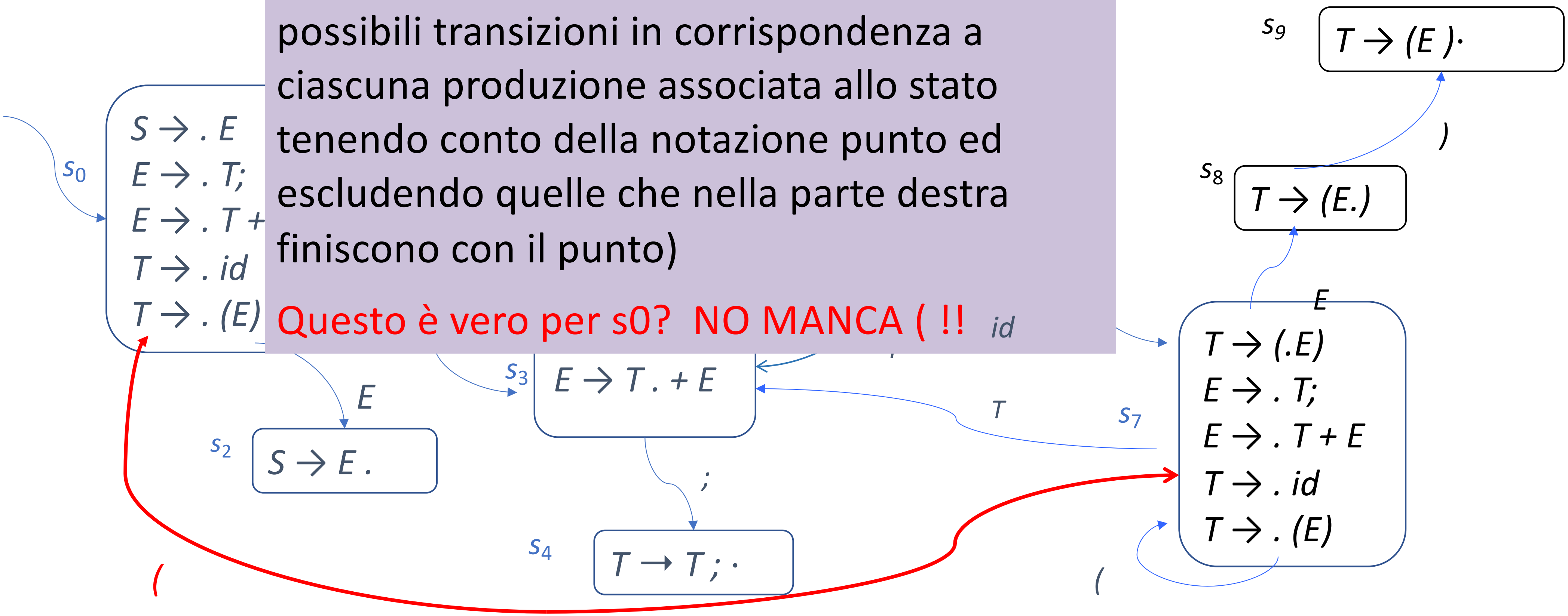
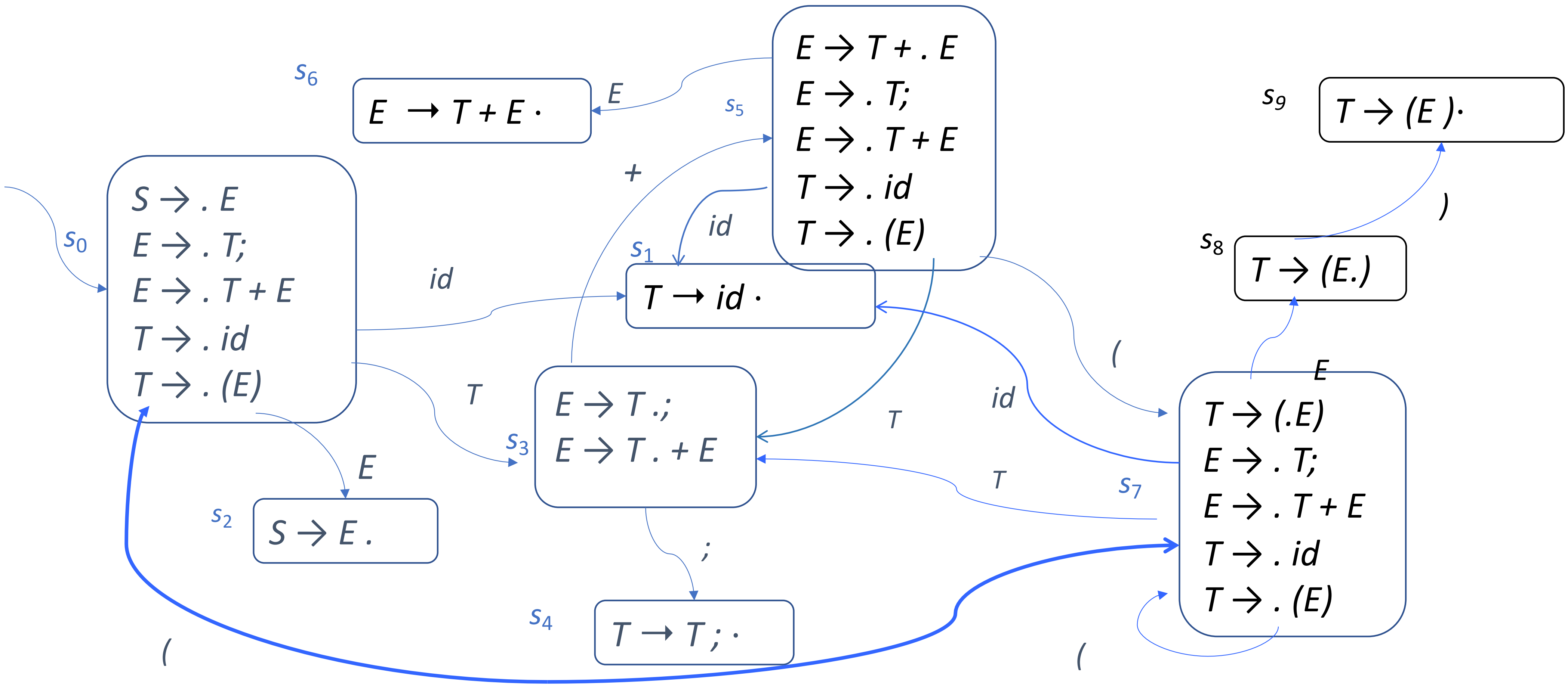


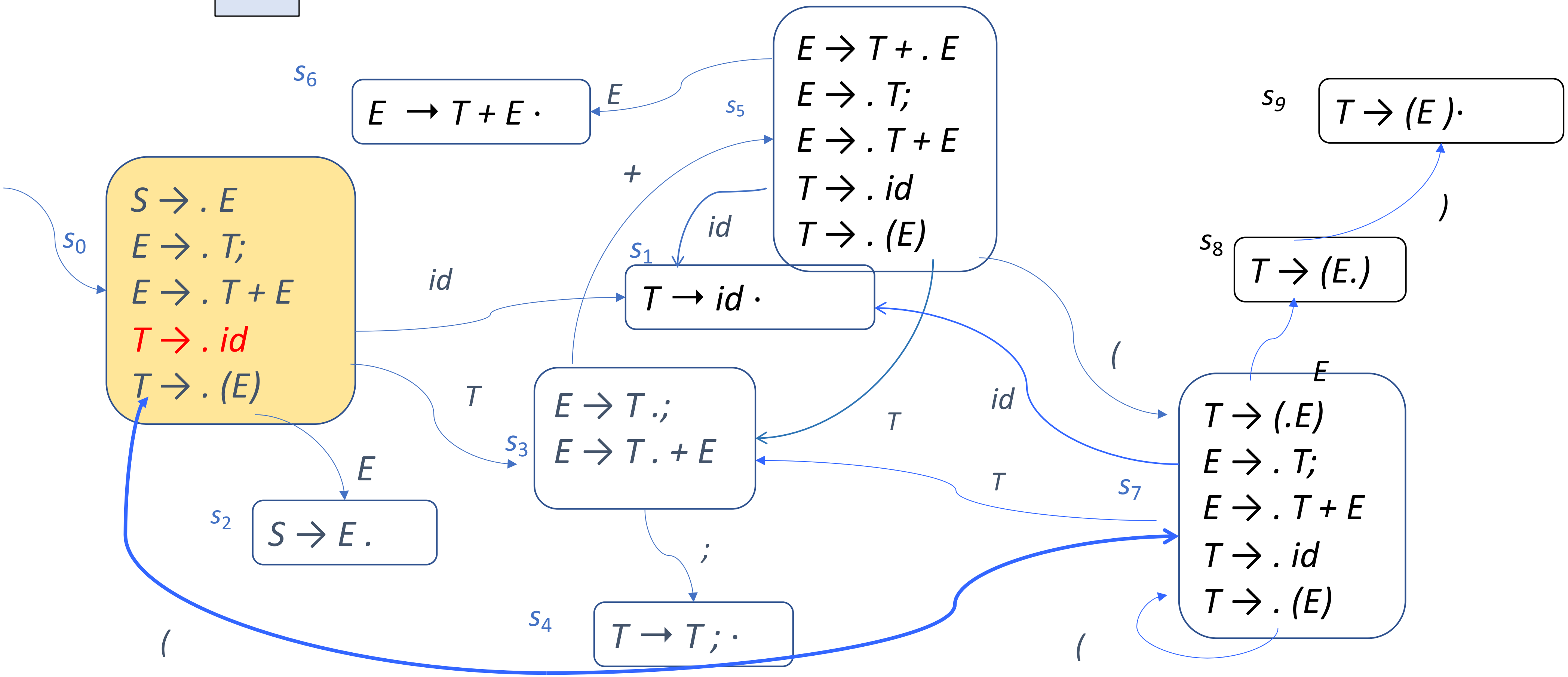
diagramma transizioni finale



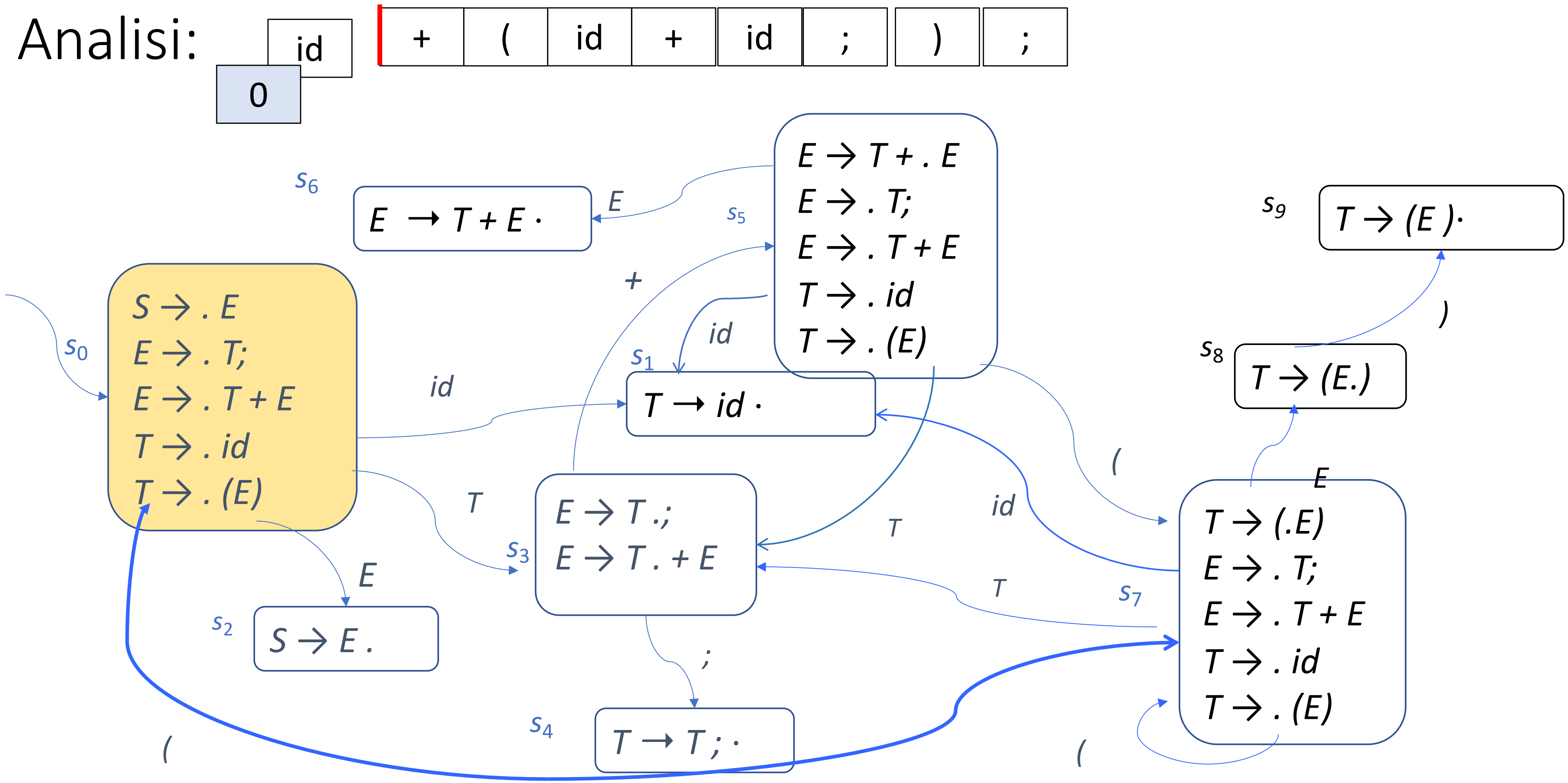
INPUT:

id	+	(	id	+	id	;	)	;

0



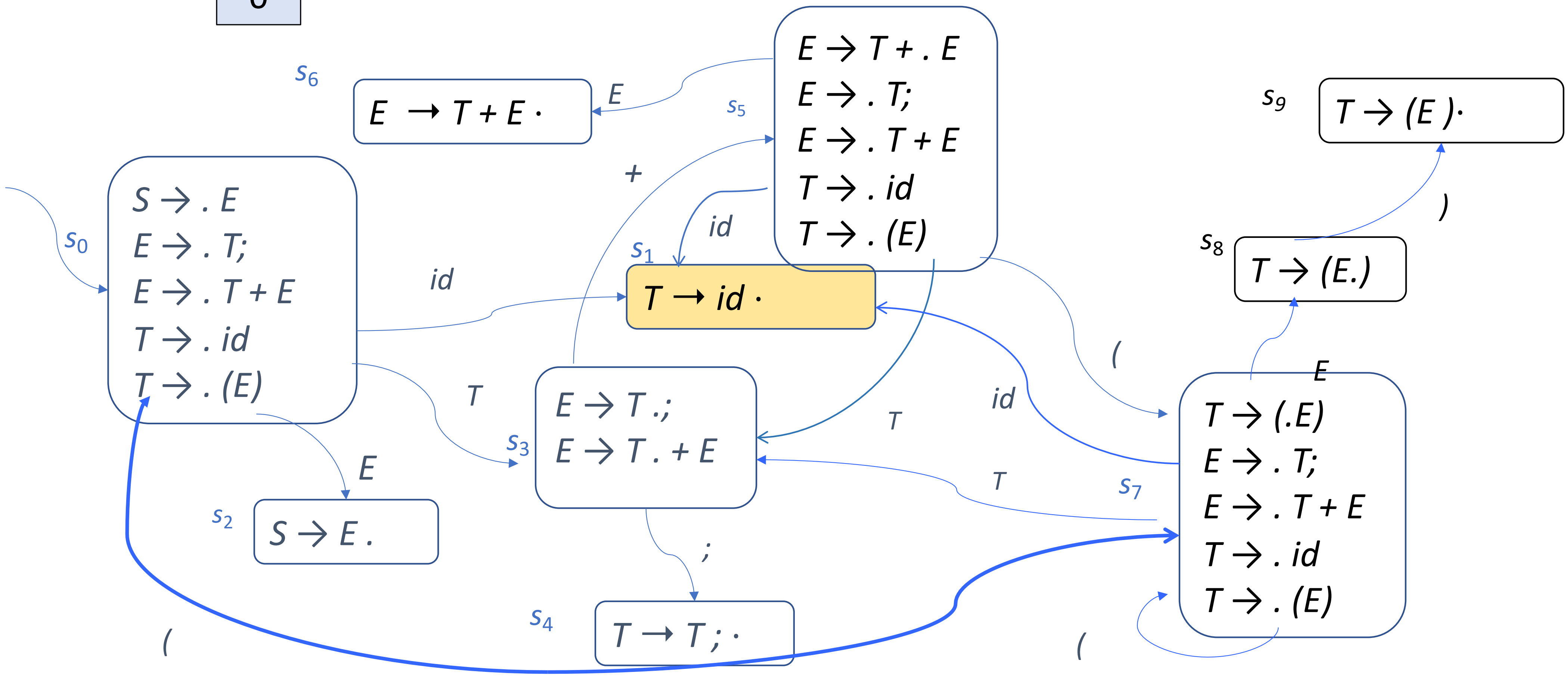
Analisi:



0
id
+ (id + id ;) ;
s_0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
s_1
T → id ·
s_2
S → E .
s_3
E → T .;
E → T . + E
s_4
T → T ; ·
s_5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s_6
E → T + E ·
s_7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
s_8
T → (E.)
s_9
T → (E)·

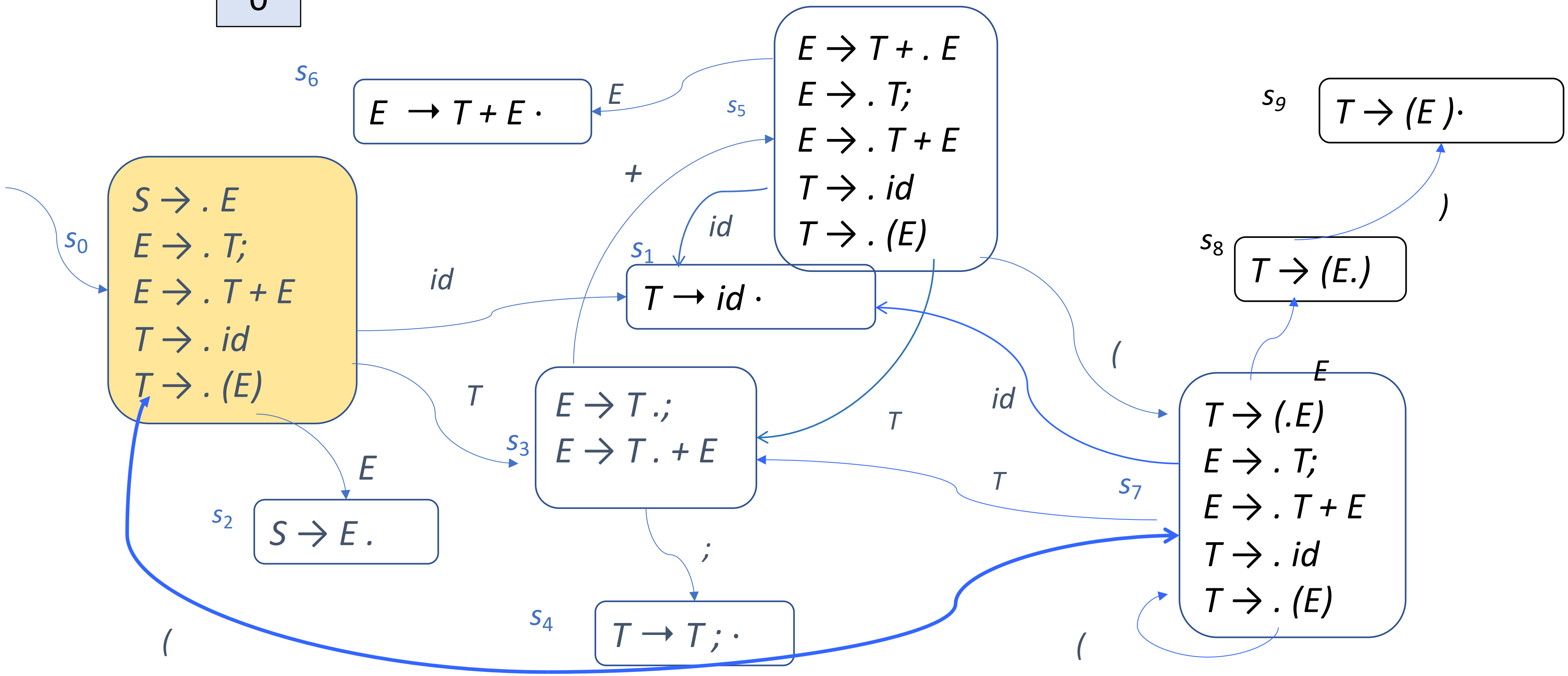
Analisi:

id	+	(	id	+	id	;	)	;
0								



Analisi:

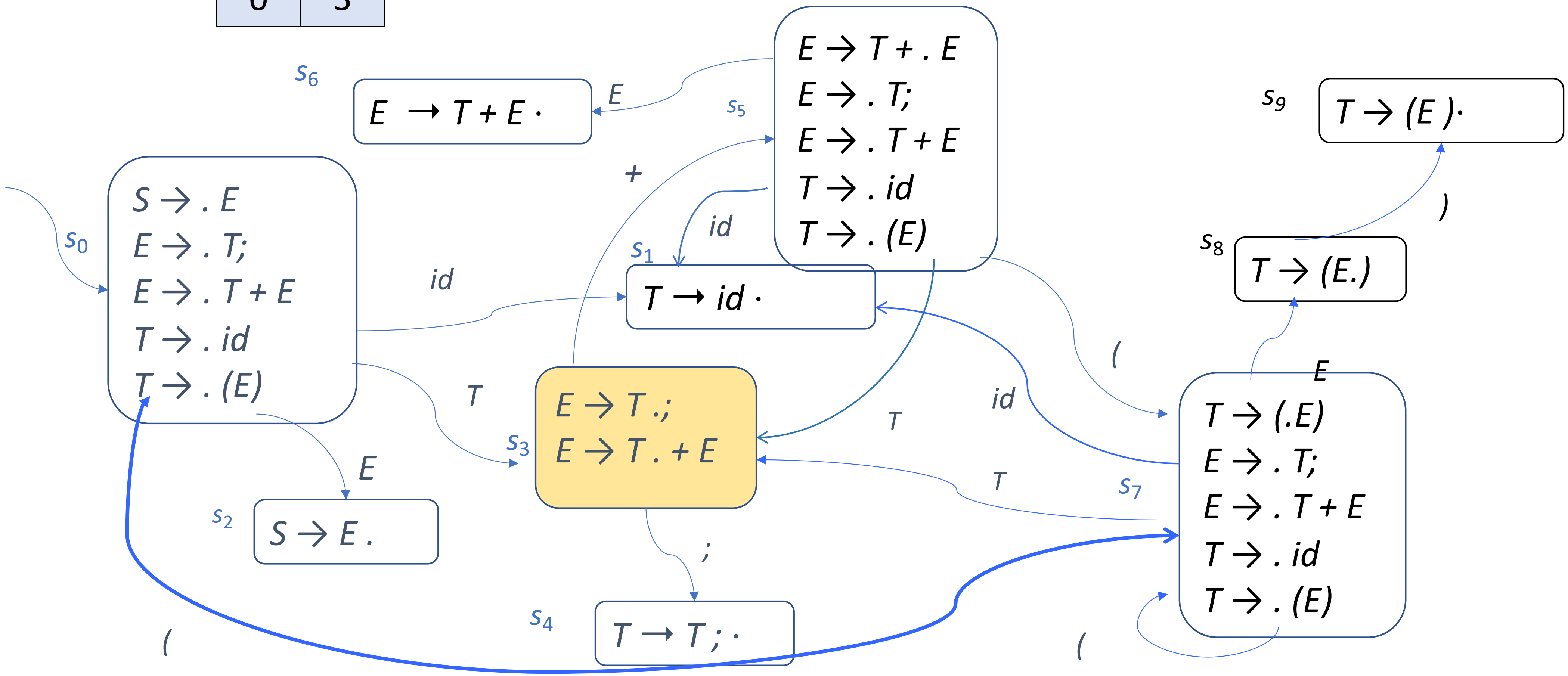
T	+	(	id	+	id	;	)	;
0								



Analisi:

T	+	(	id	+	id	;	)	;

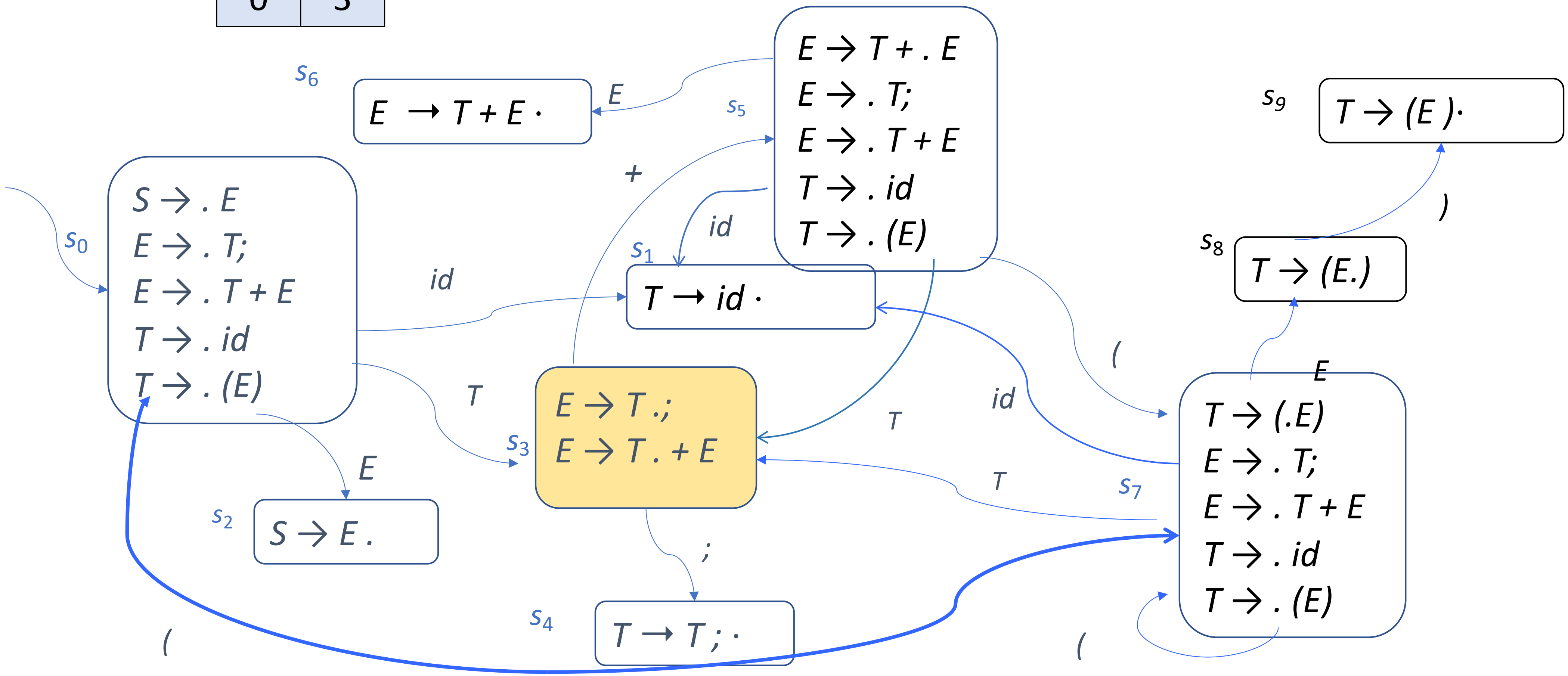
0	3



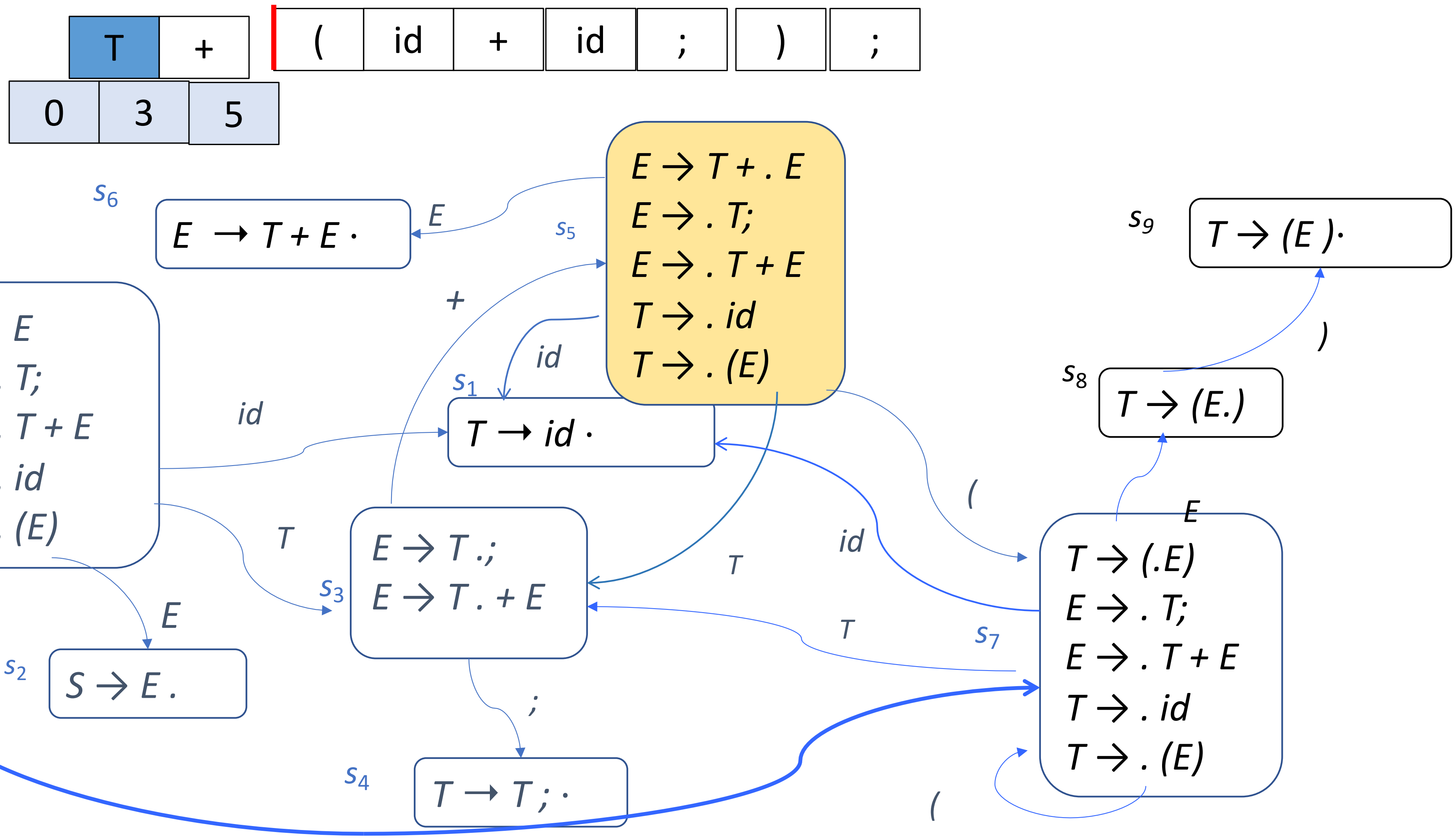
Analisi:

T	+
0	3

(	id	+	id	;	)	;

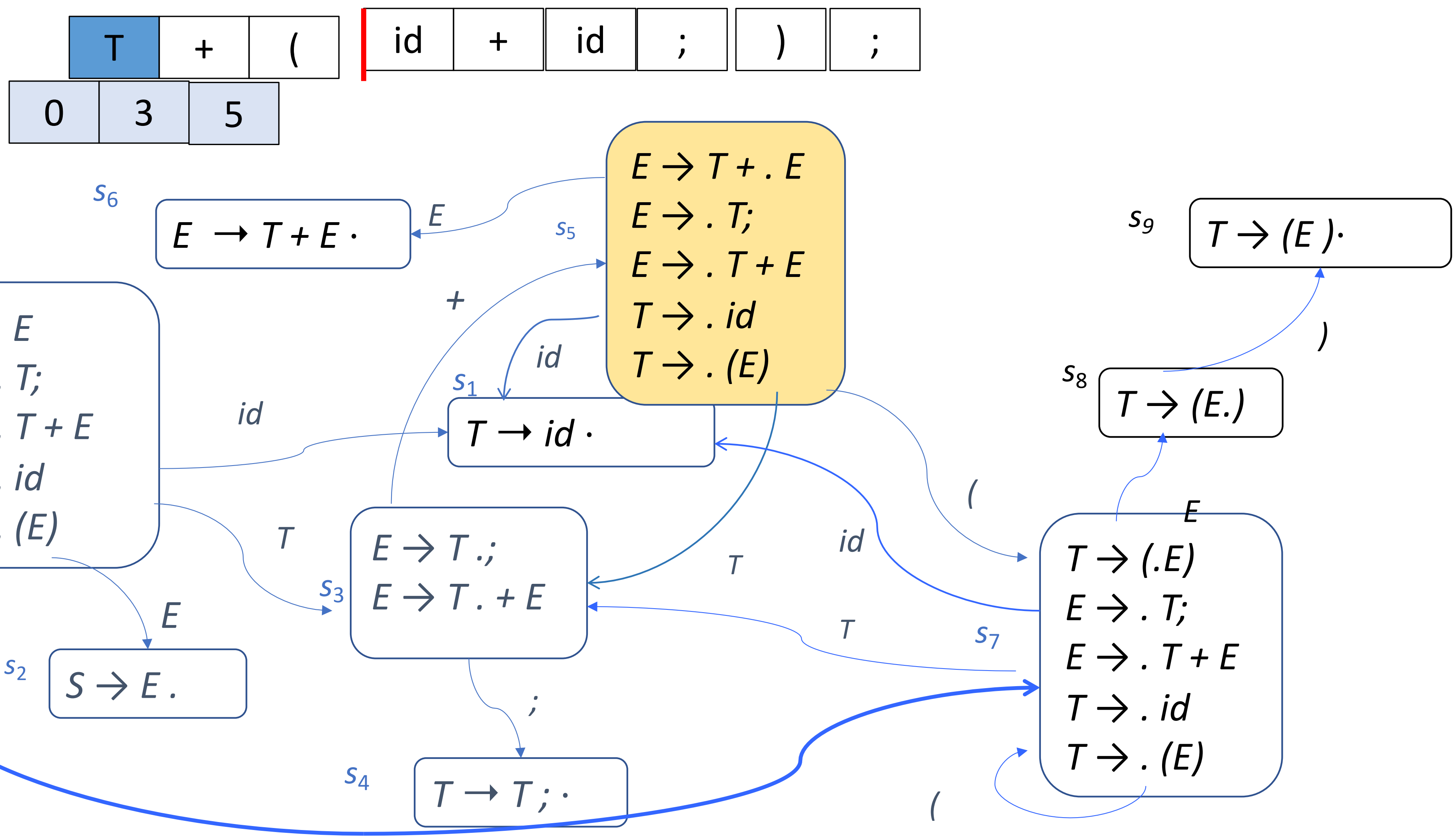


Analisi:

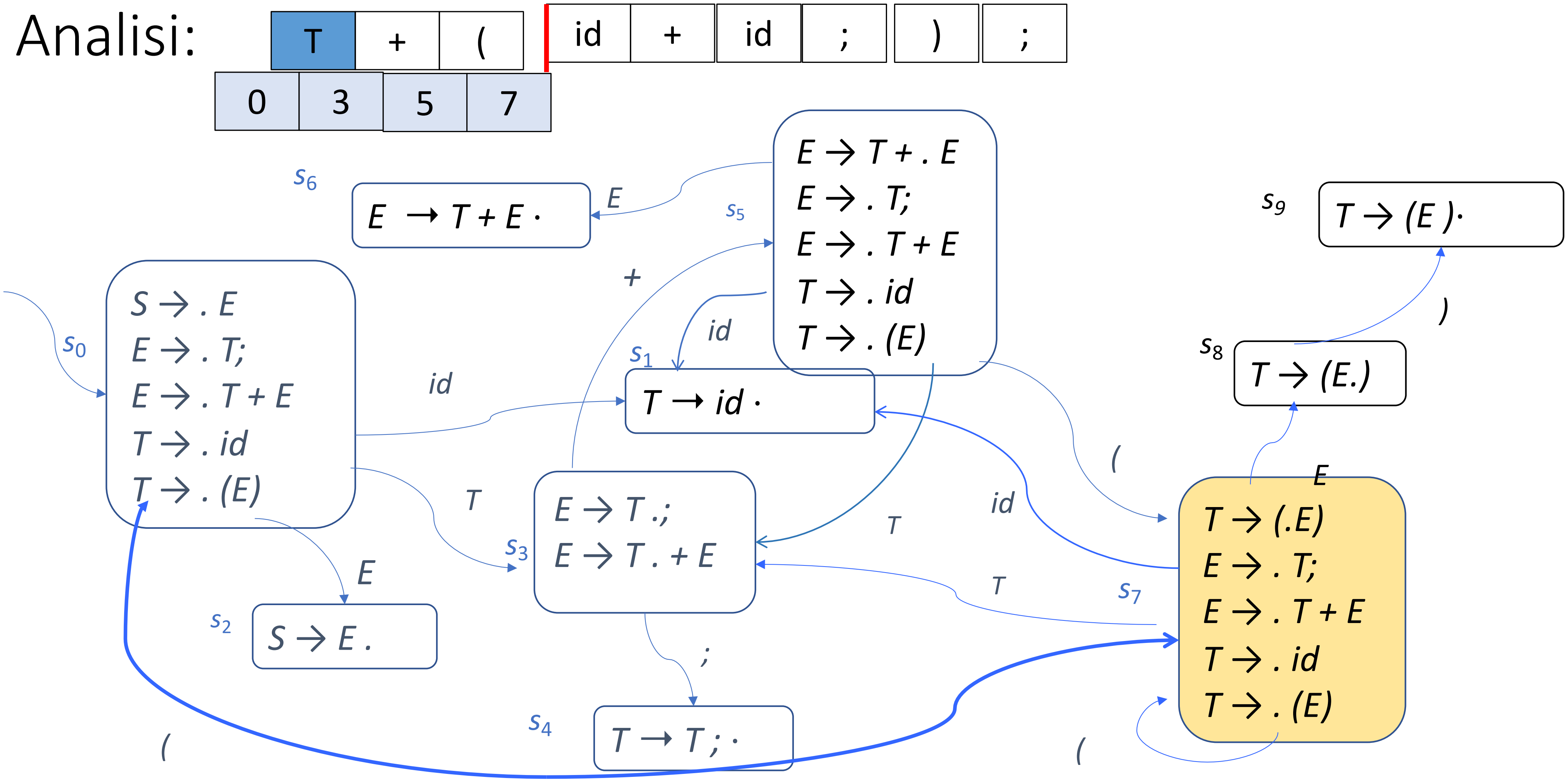


T
+
(
id
+
id
;
)
;
0
3
5
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
s1
T → id ·
s2
S → E .
s3
E → T .;
E → T . + E
s4
T → T ; ·
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s6
E → T + E ·
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
s8
T → (E.)
s9
T → (E)·
id
T
E
+
;
(
)

Analisi:

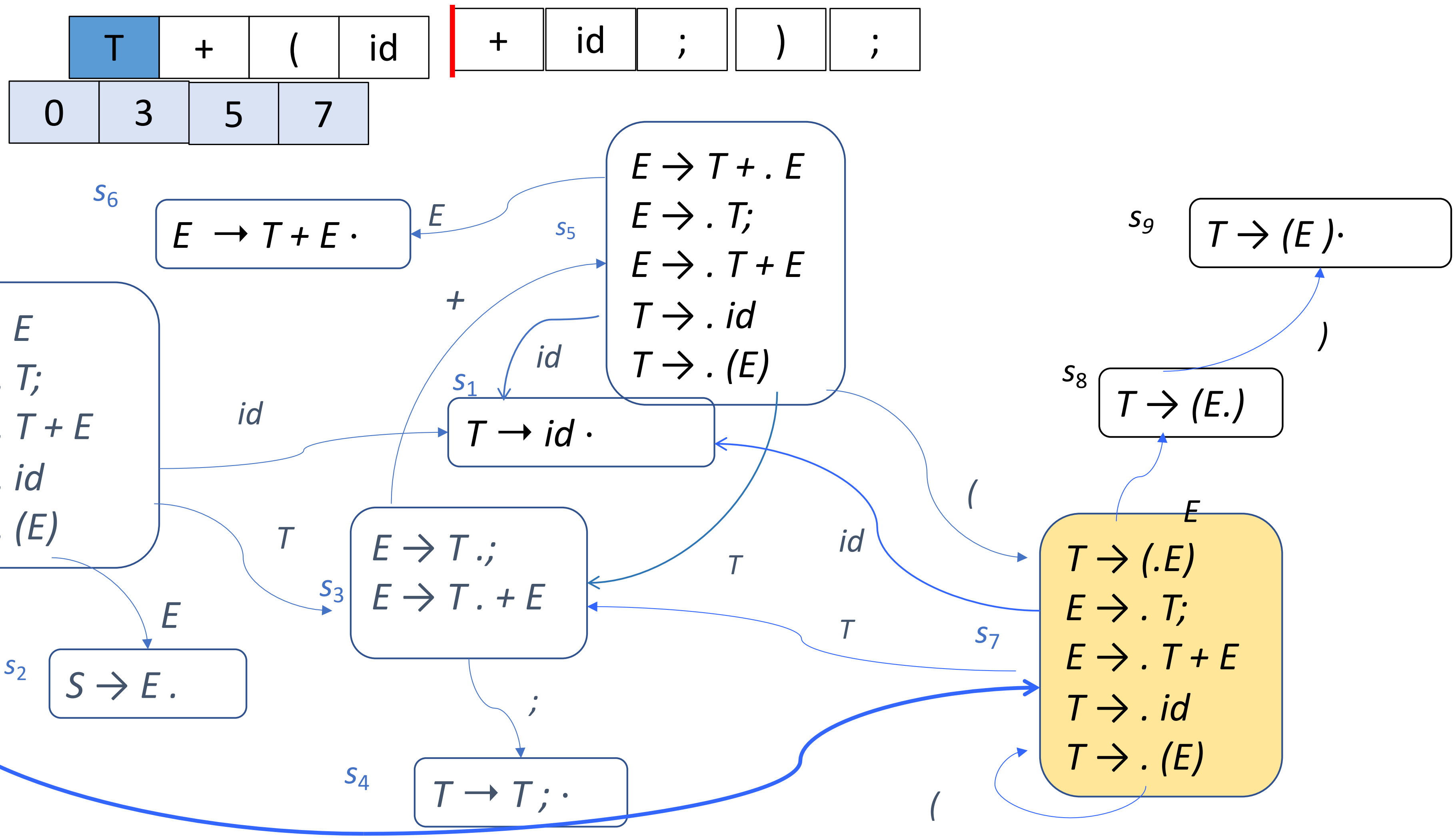

T + (
id + id ;) ;
0 3 5
s6
E → T + E ·
E
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s9
T → (E)·
+
id
s1
T → id ·
)
s8
T → (E.)
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
id
(
T
T
id
E
s3
E → T .;
E → T . + E
T
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
E
s2
S → E .
;
s4
T → T ; ·
(
(

Analisi:



T + (
id + id ;) ;
0 3 5 7
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
s1
T → id ·
s2
S → E .
s3
E → T .;
E → T . + E
s4
T → T ; ·
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s6
E → T + E ·
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
s8
T → (E.)
s9
T → (E)·
id
T
E
+
;
(
)

Analisi:

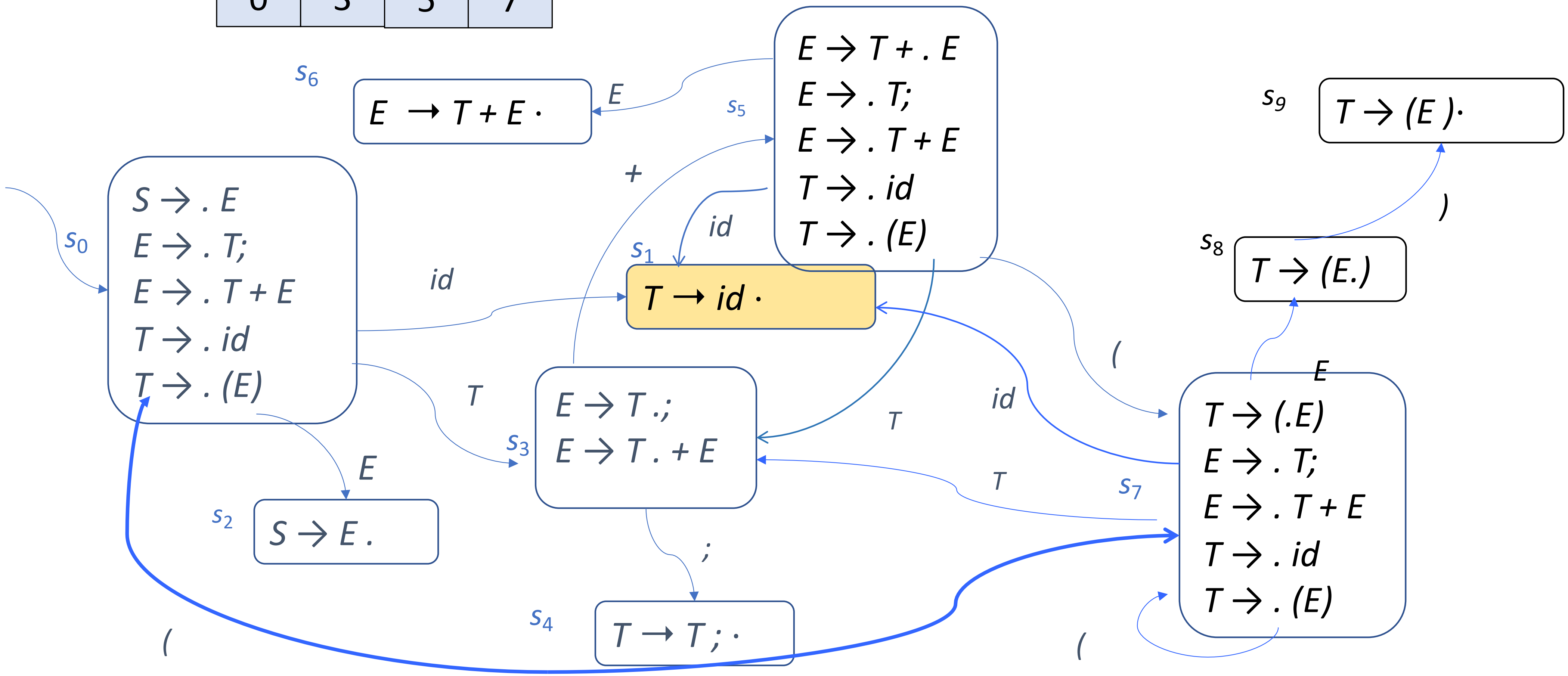


T + (id
+ id ;) ;
0 3 5 7
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
s6
E → T + E ·
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s1
T → id ·
s2
S → E .
s3
E → T .;
E → T . + E
s4
T → T ; ·
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
s8
T → (E.)
s9
T → (E)·
id
T
E
+
id
E
;
T
T
id
(
(
(
E
)

Analisi:

	T	+	(
0	3	5	7

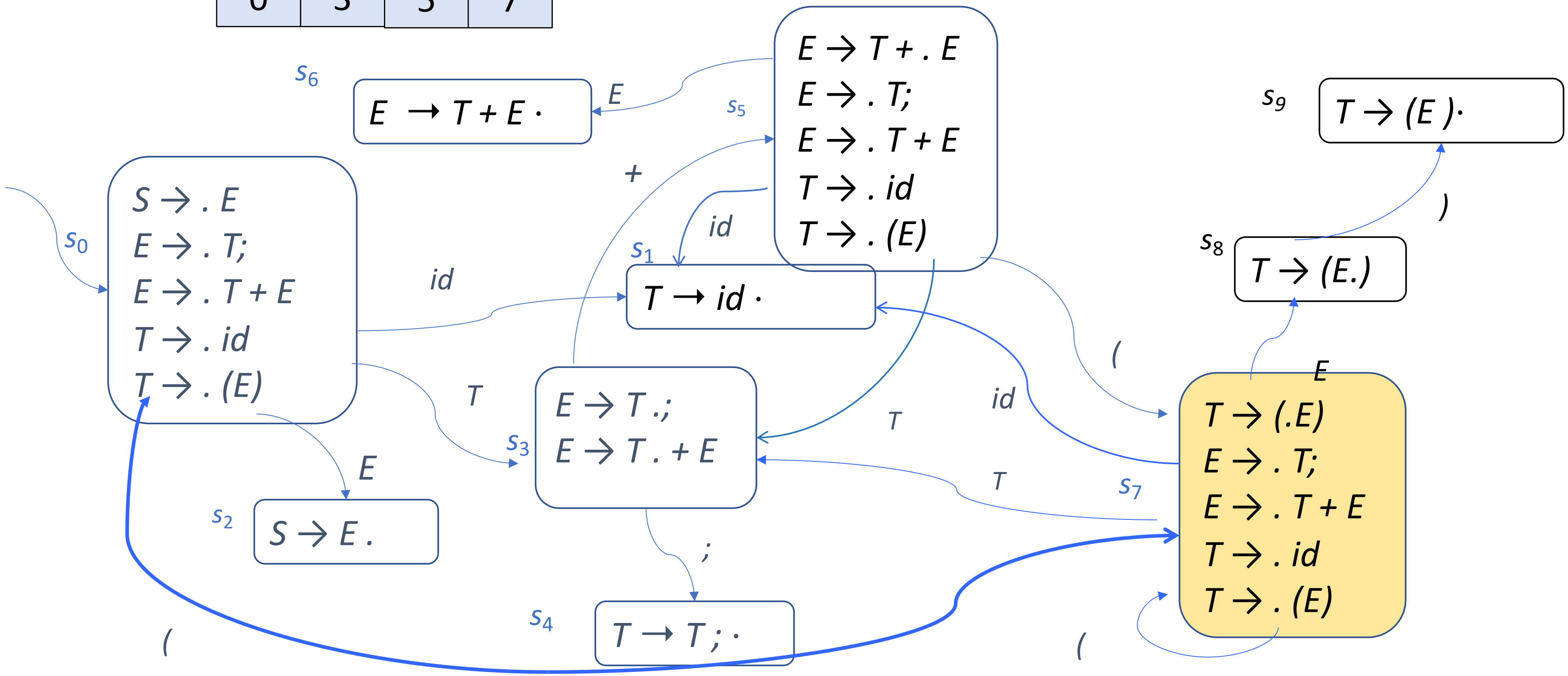
+	id	;	)	;



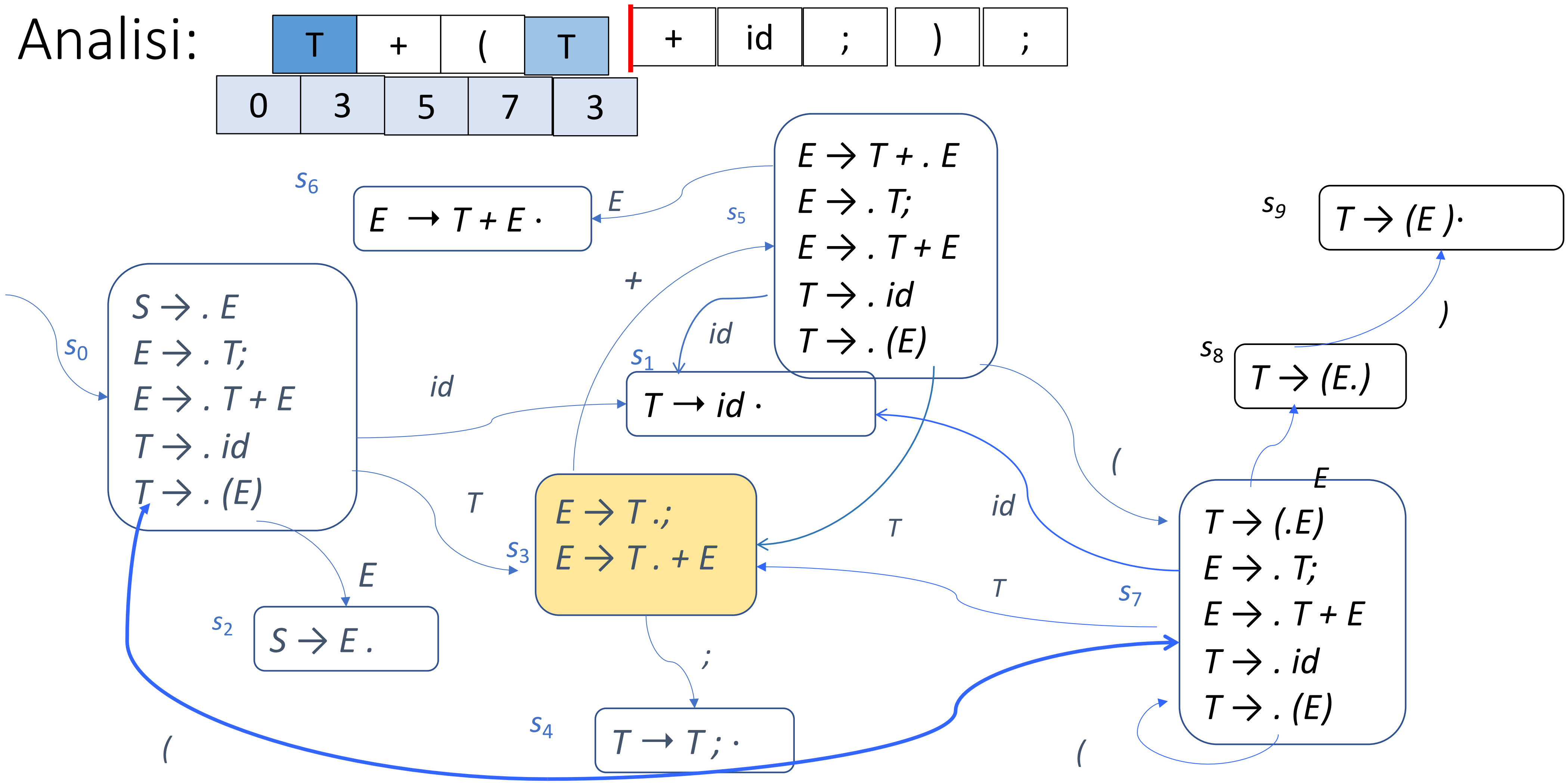
Analisi:

T	+	(	T	+	id	;	)	;

0	3	5	7



Analisi:

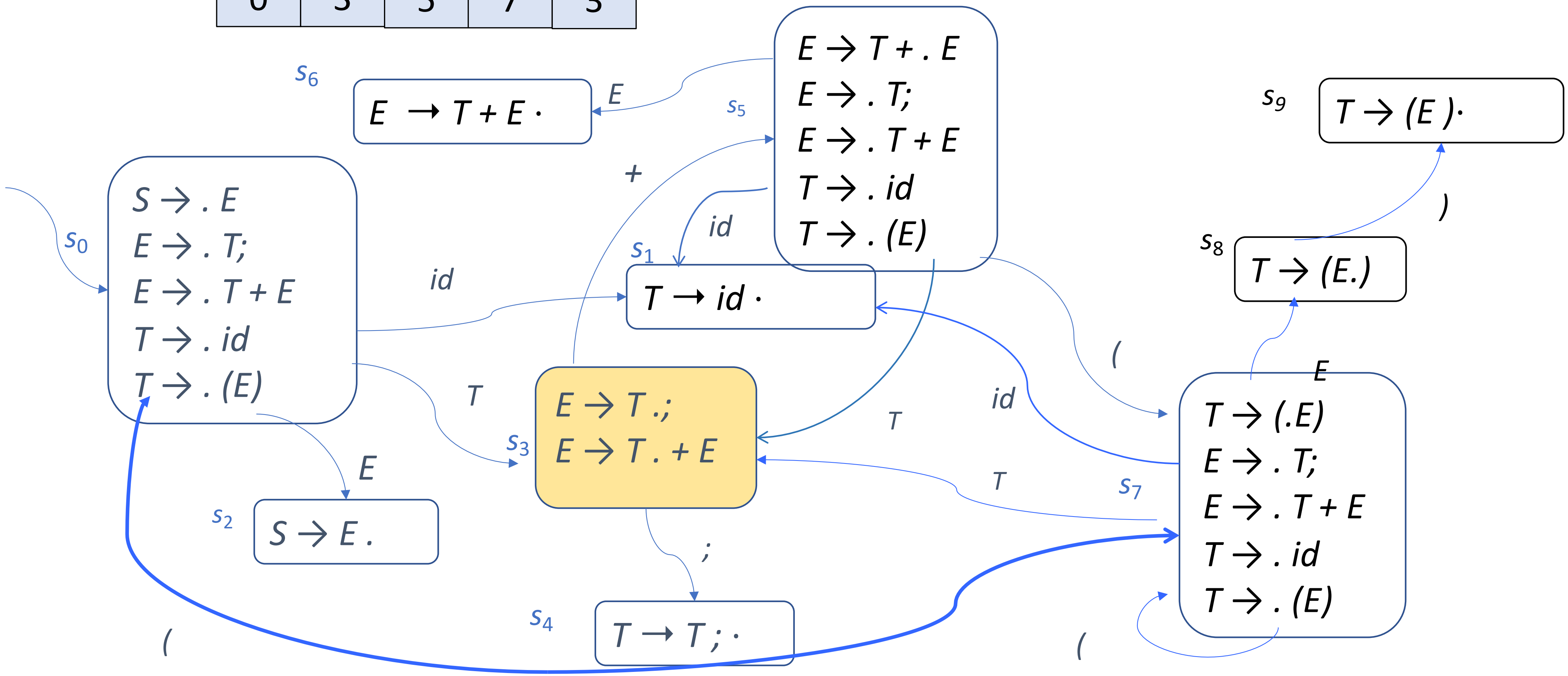


T + (T
+ id ;) ;
0 3 5 7 3
s_0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
s_1
T → id ·
s_2
S → E .
s_3
E → T .;
E → T . + E
s_4
T → T ; ·
s_5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s_6
E → T + E ·
s_7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
s_8
T → (E.)
s_9
T → (E)·
id
T
E
+
id
;
(
T
T
id
(
E
)
(

Analisi:

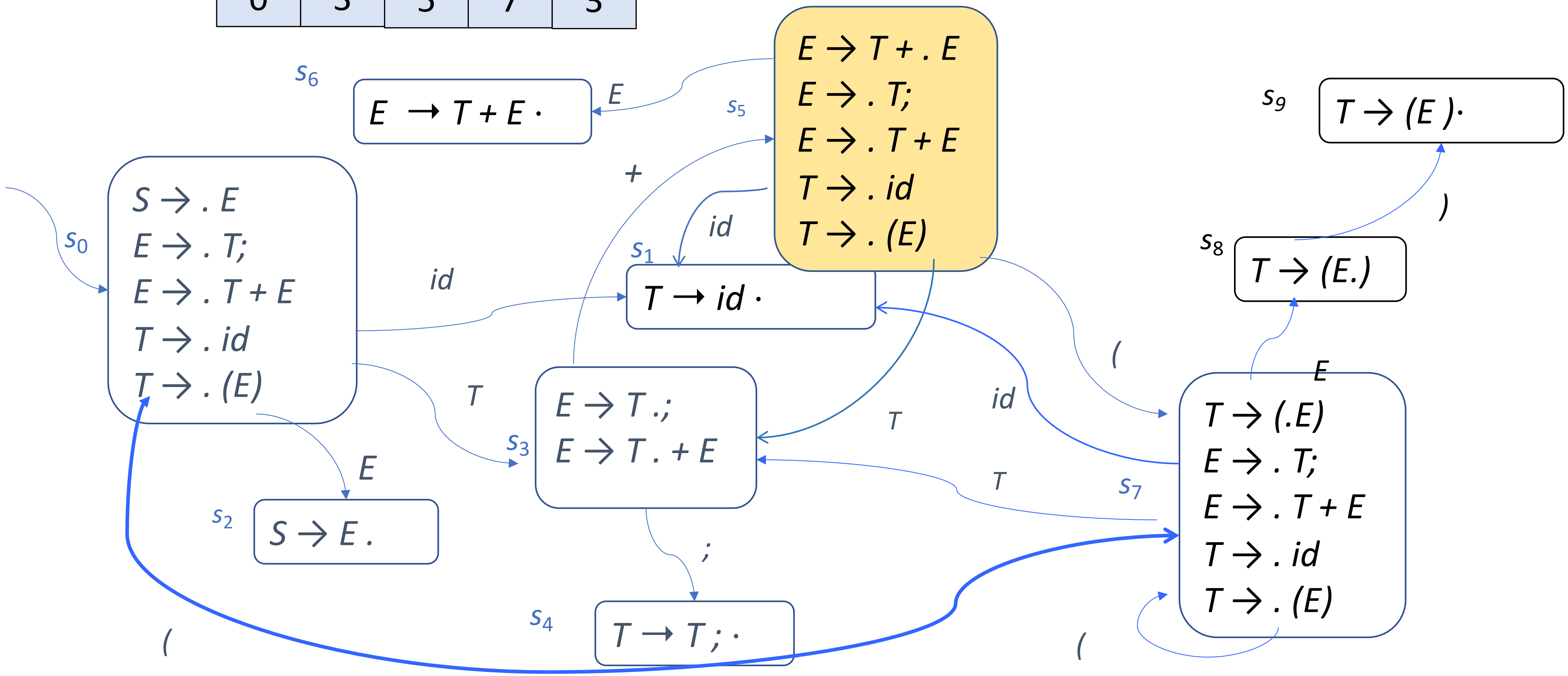
T	+	(	T	+	id	;	)	;

0	3	5	7	3



Analisi:

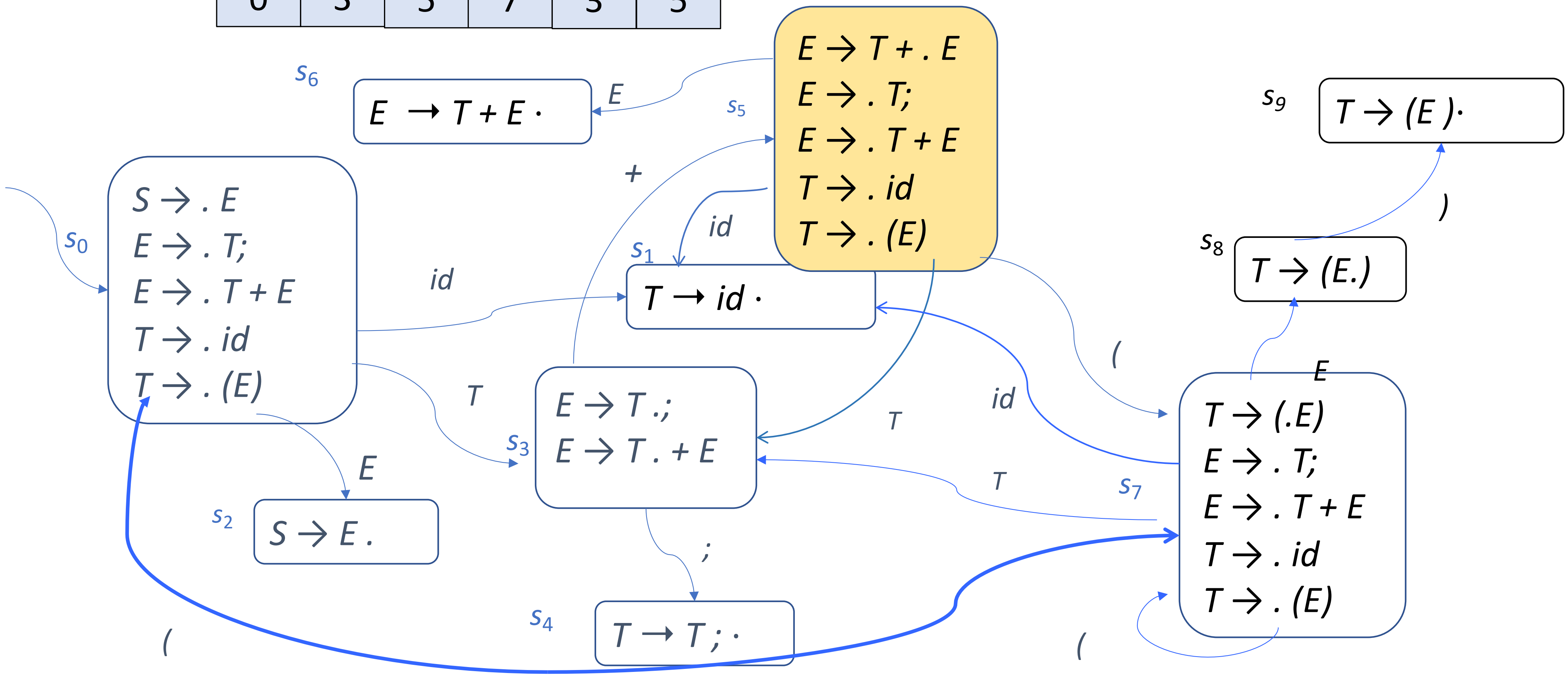
T	+	(	T	+	id	;	)	;
0	3	5	7	3				



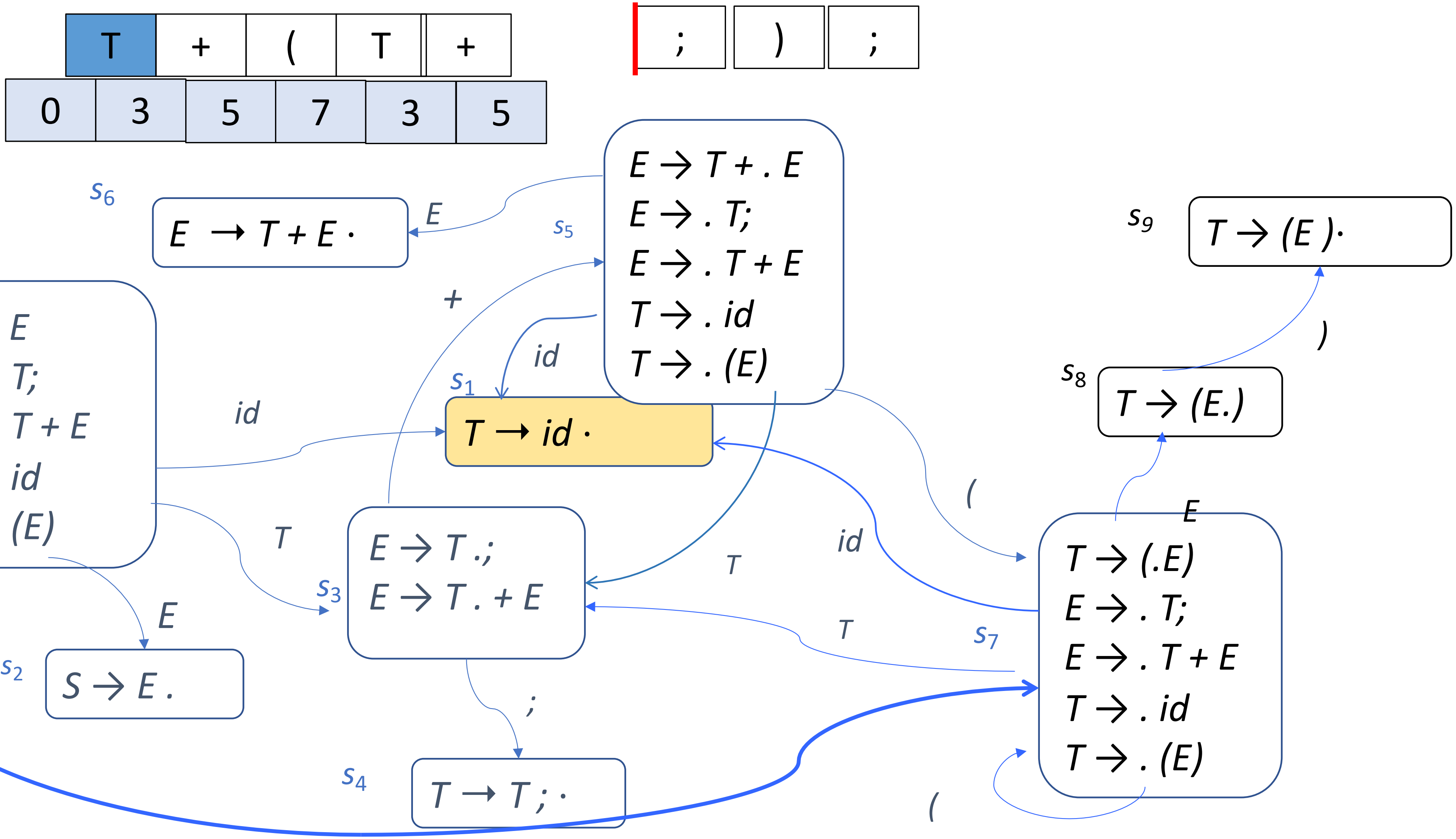
Analisi:

T	+	(	T	+	id	;	)	;

0	3	5	7	3	5

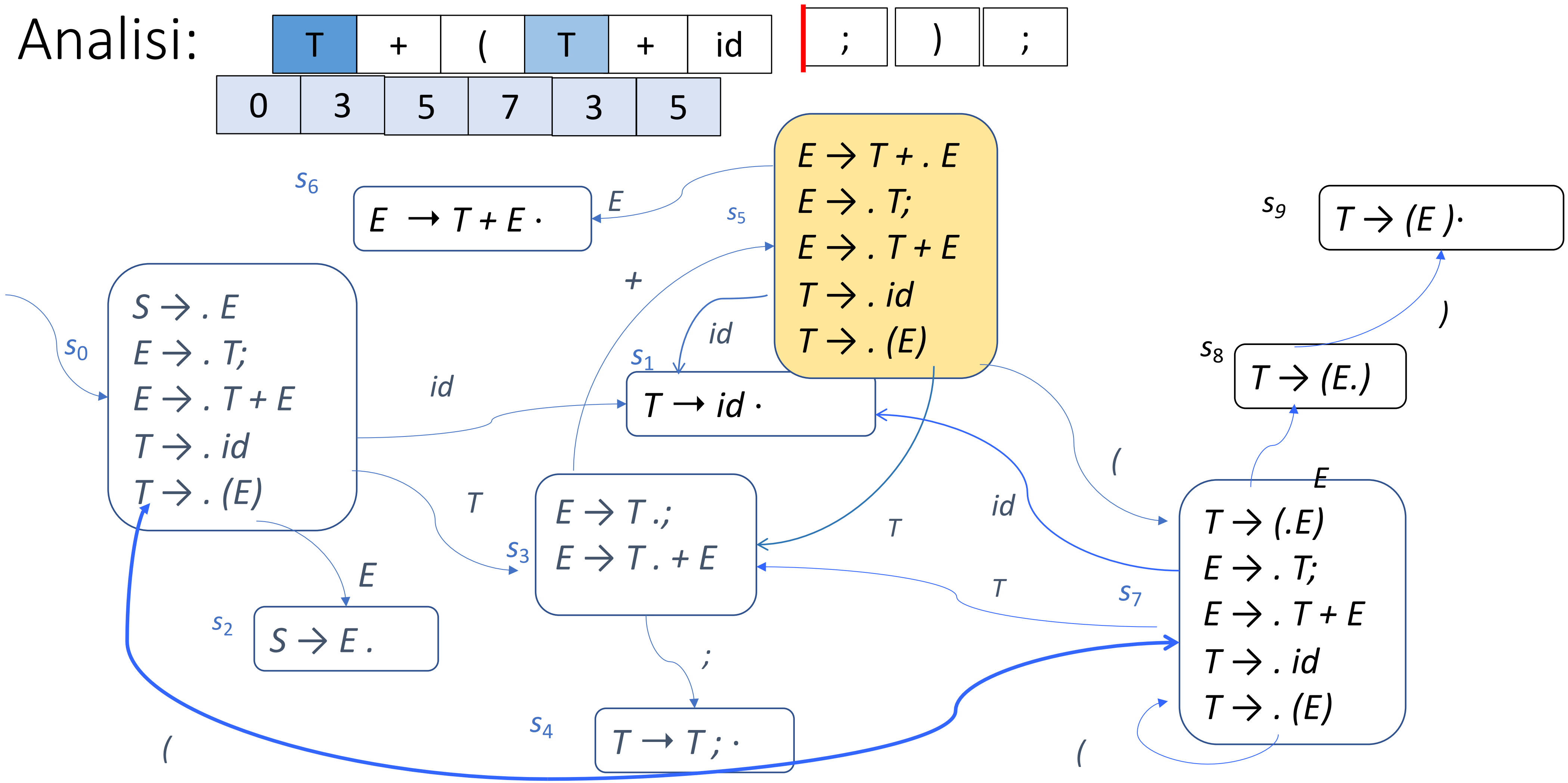


Analisi:



T + (T +
0 3 5 7 3 5
;) ;
s6
E → T + E ·
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s9
T → (E)·
s8
T → (E.)
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
s1
T → id ·
s3
E → T .;
E → T . + E
s2
S → E .
s4
T → T ; ·
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
id
+
id
E
T
E
;
(
T
id
T
(
(
)
E

Analisi:

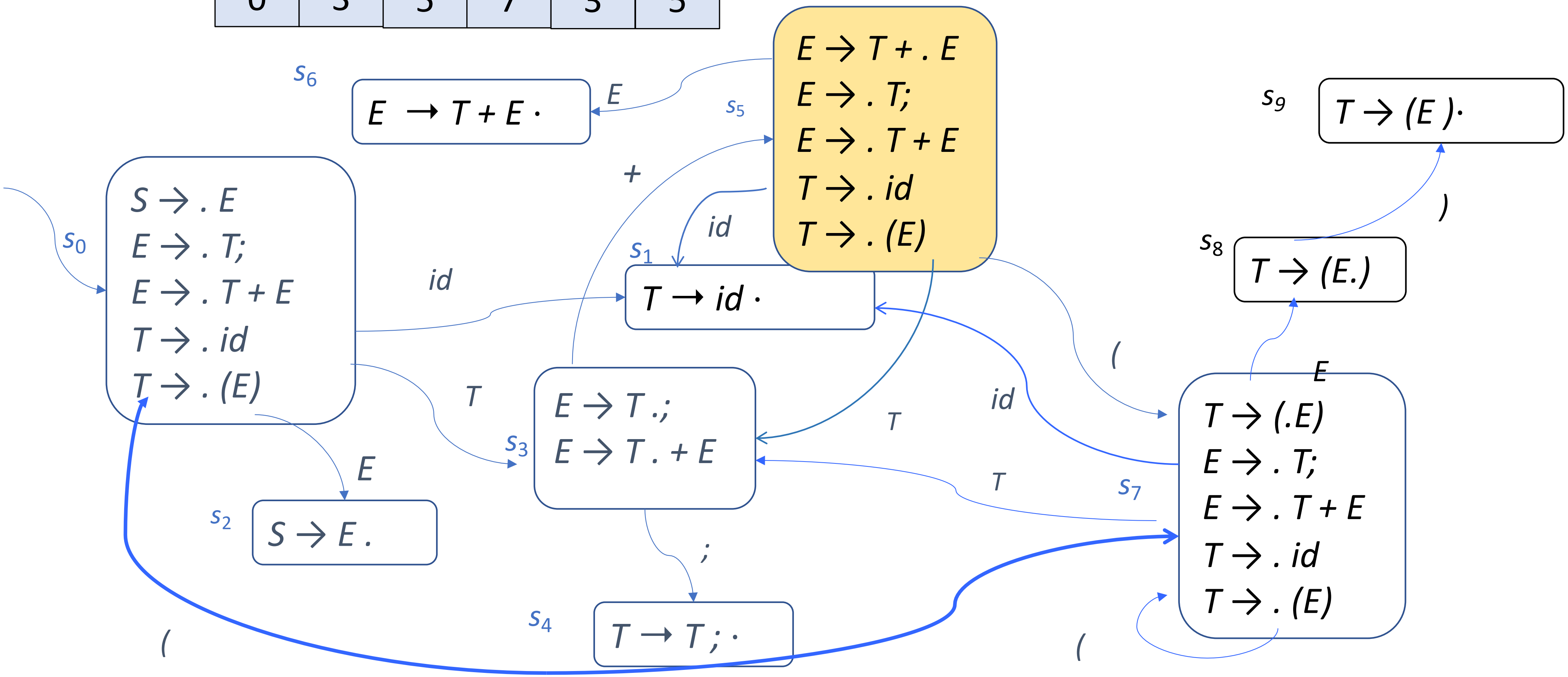


T
+
(
T
+
id
;
)
;
0
3
5
7
3
5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s6
E → T + E ·
E
s5
s9
T → (E)·
+
)
id
s8
T → (E.)
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
id
s1
T → id ·
(
T
E
E → T .;
E → T . + E
T
id
s3
T
s7
E
s2
S → E .
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
;
s4
T → T ; ·
(
(

Analisi:

	T	+	(	T	+
0	3	5	7	3	5

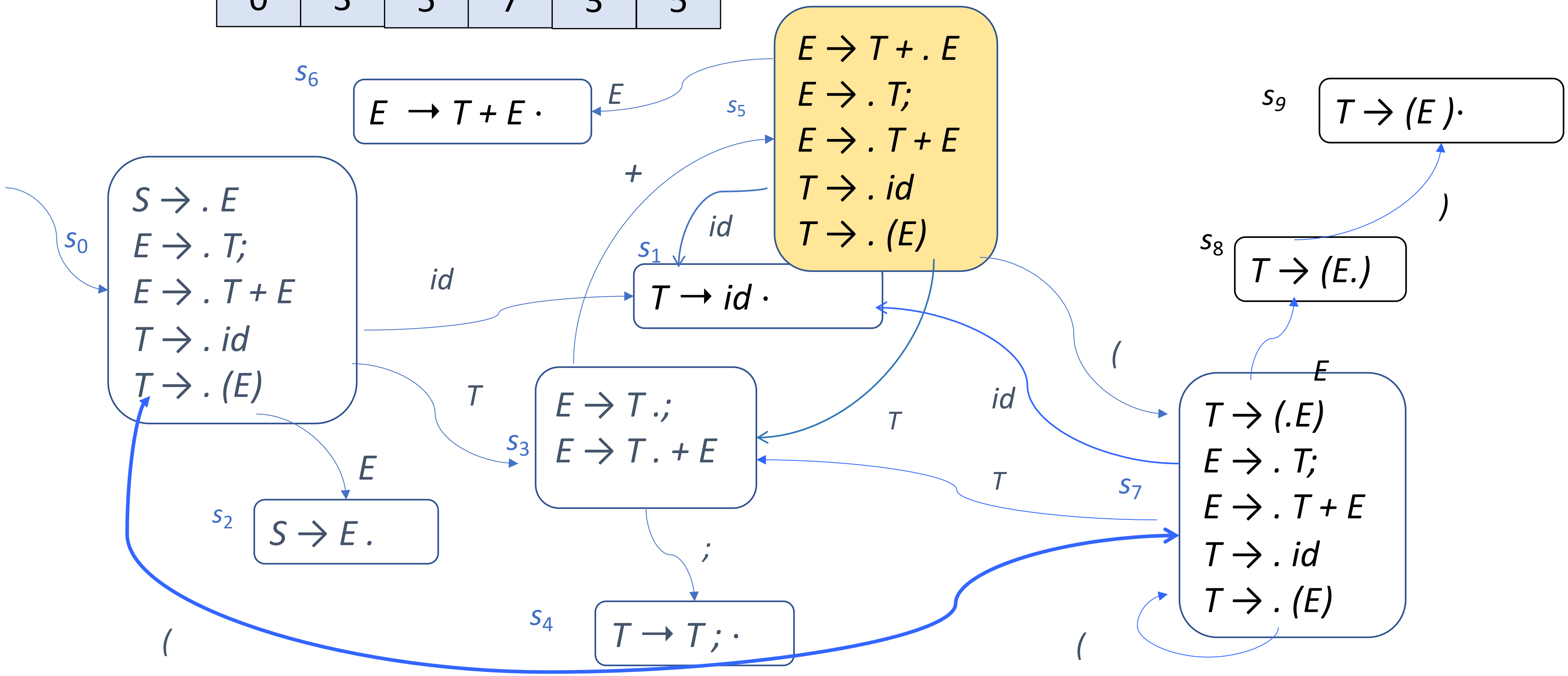
;	)	;



Analisi:

T	+	(	T	+	T	;	)	;

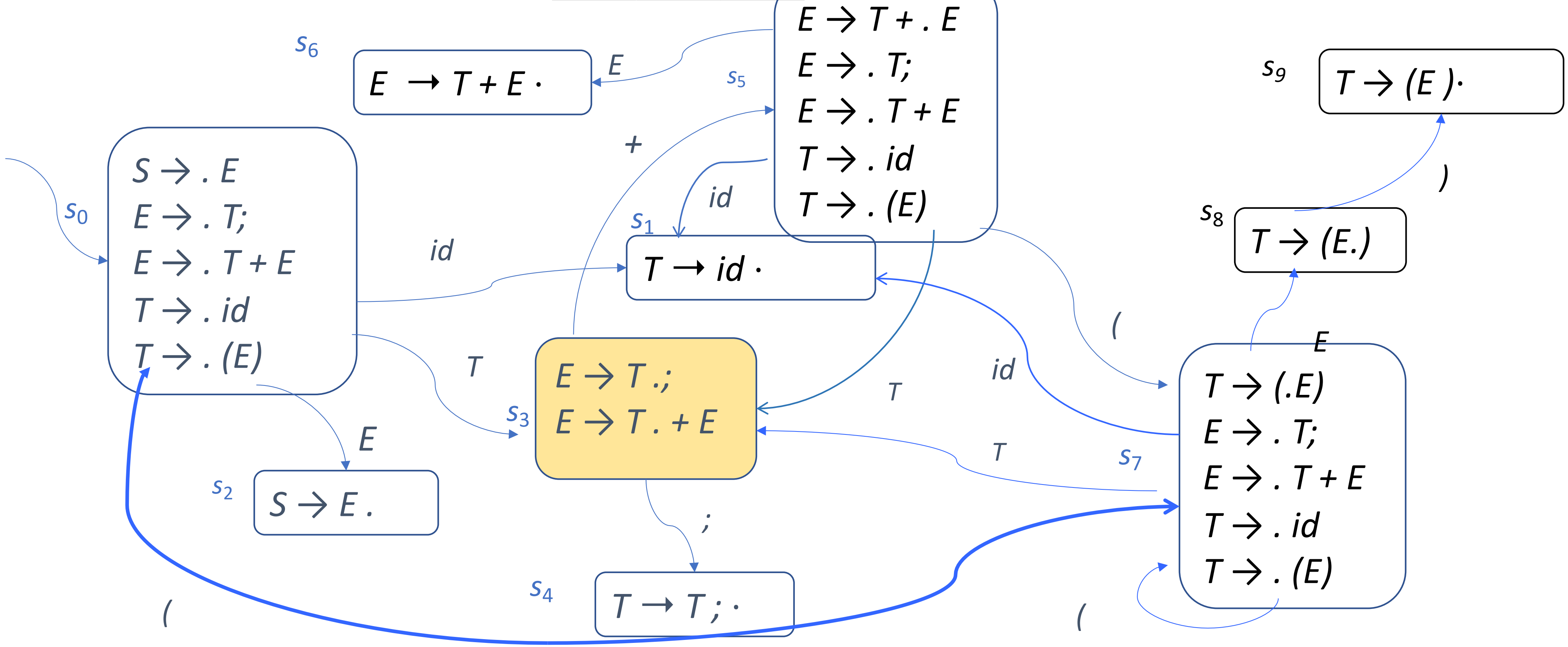
0	3	5	7	3	5



Analisi:

T	+	(	T	+	T	;	)	;

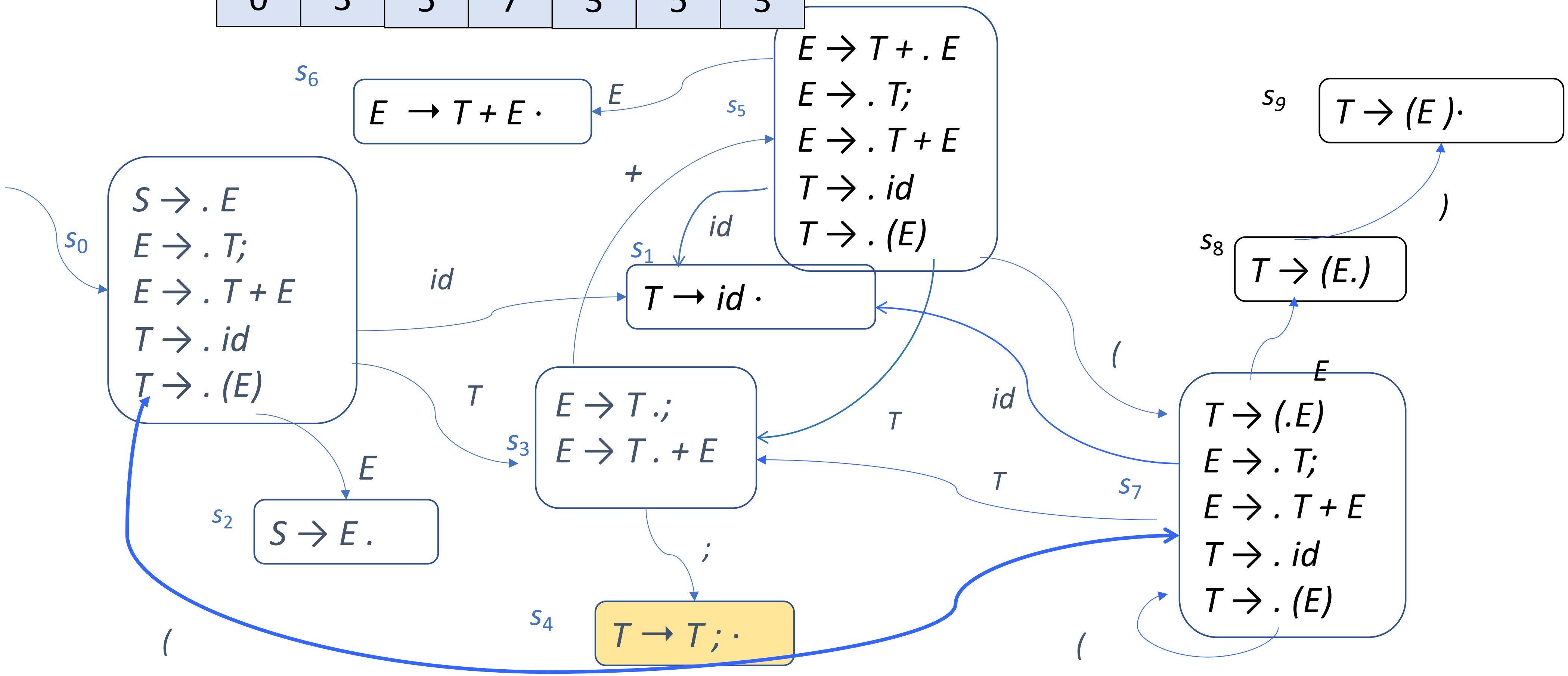
0	3	5	7	3	5	3



Analisi:

T	+	(	T	+	T	;		)	;

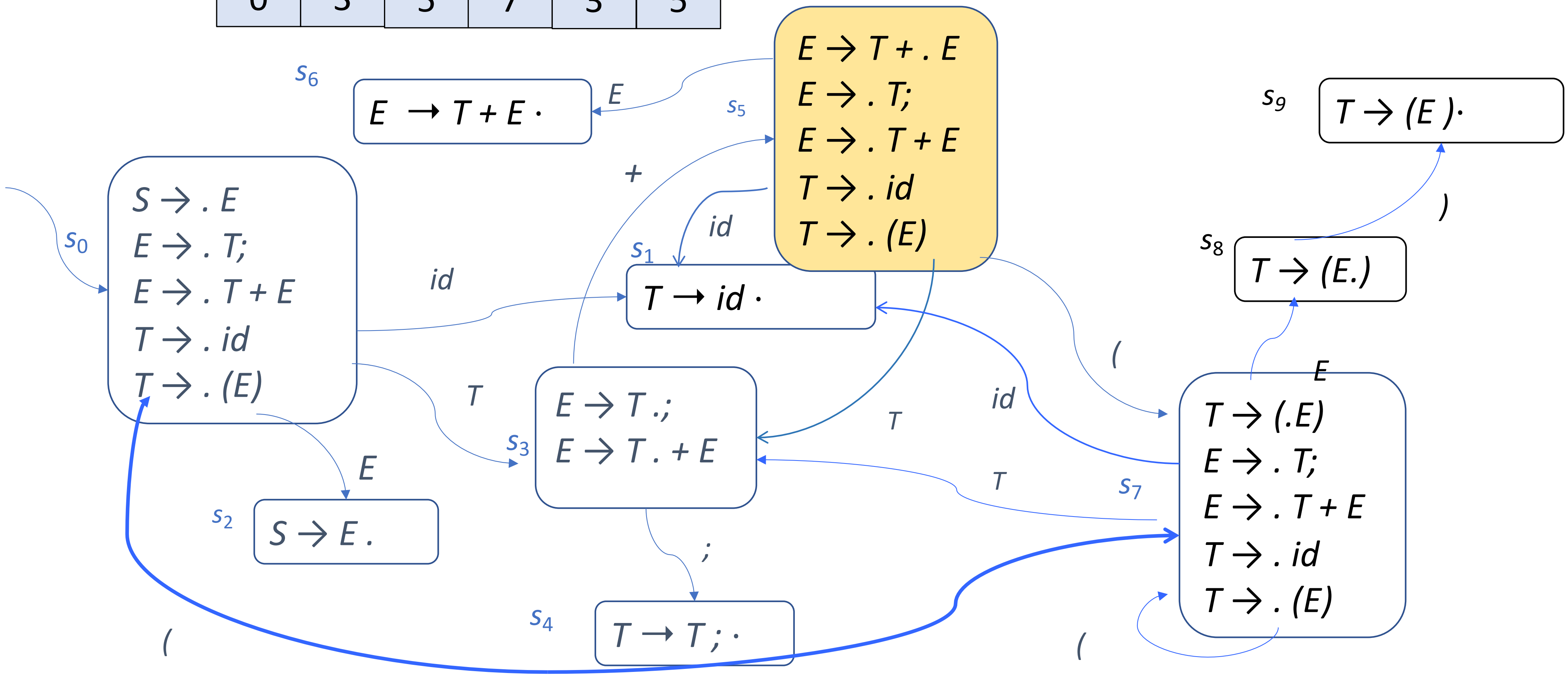
0	3	5	7	3	5	3



Analisi:

T	+	(	T	+	
0	3	5	7	3	5

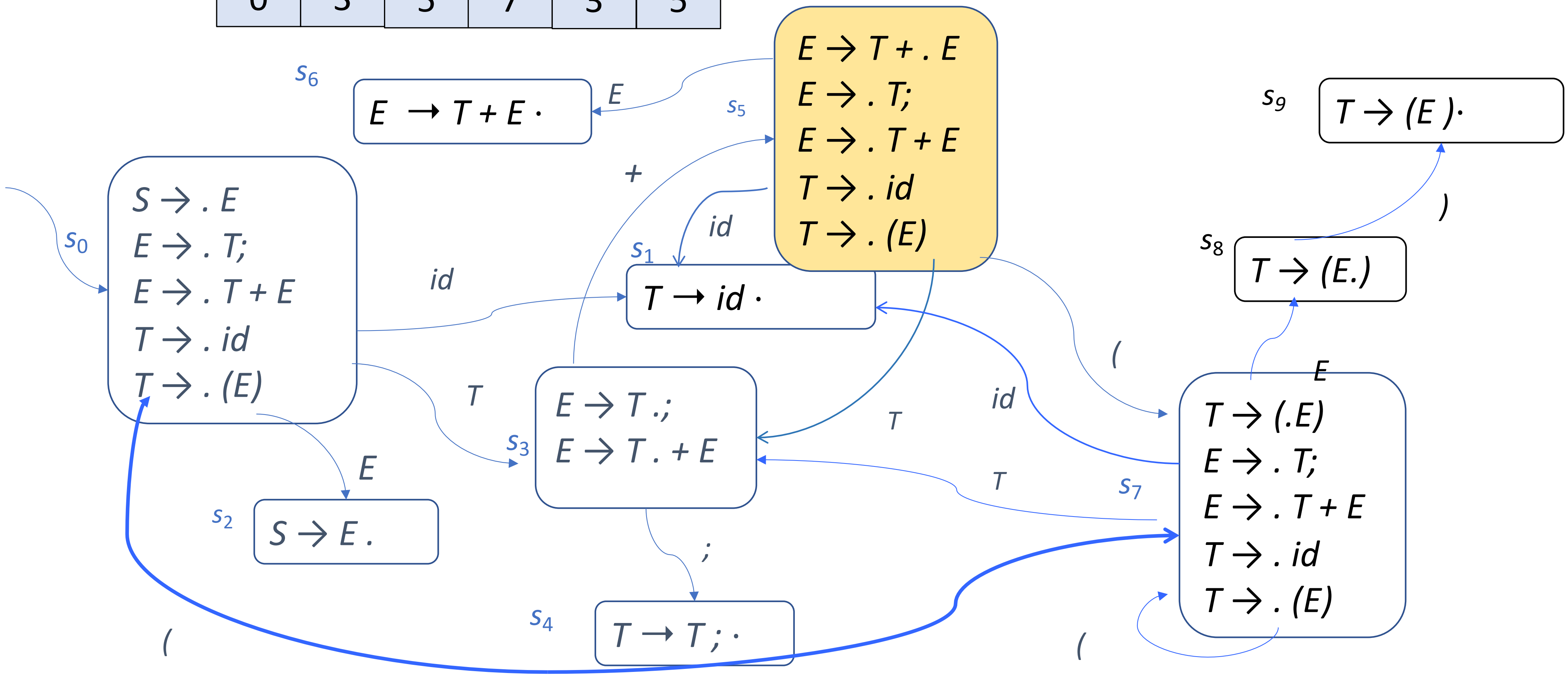
)	;



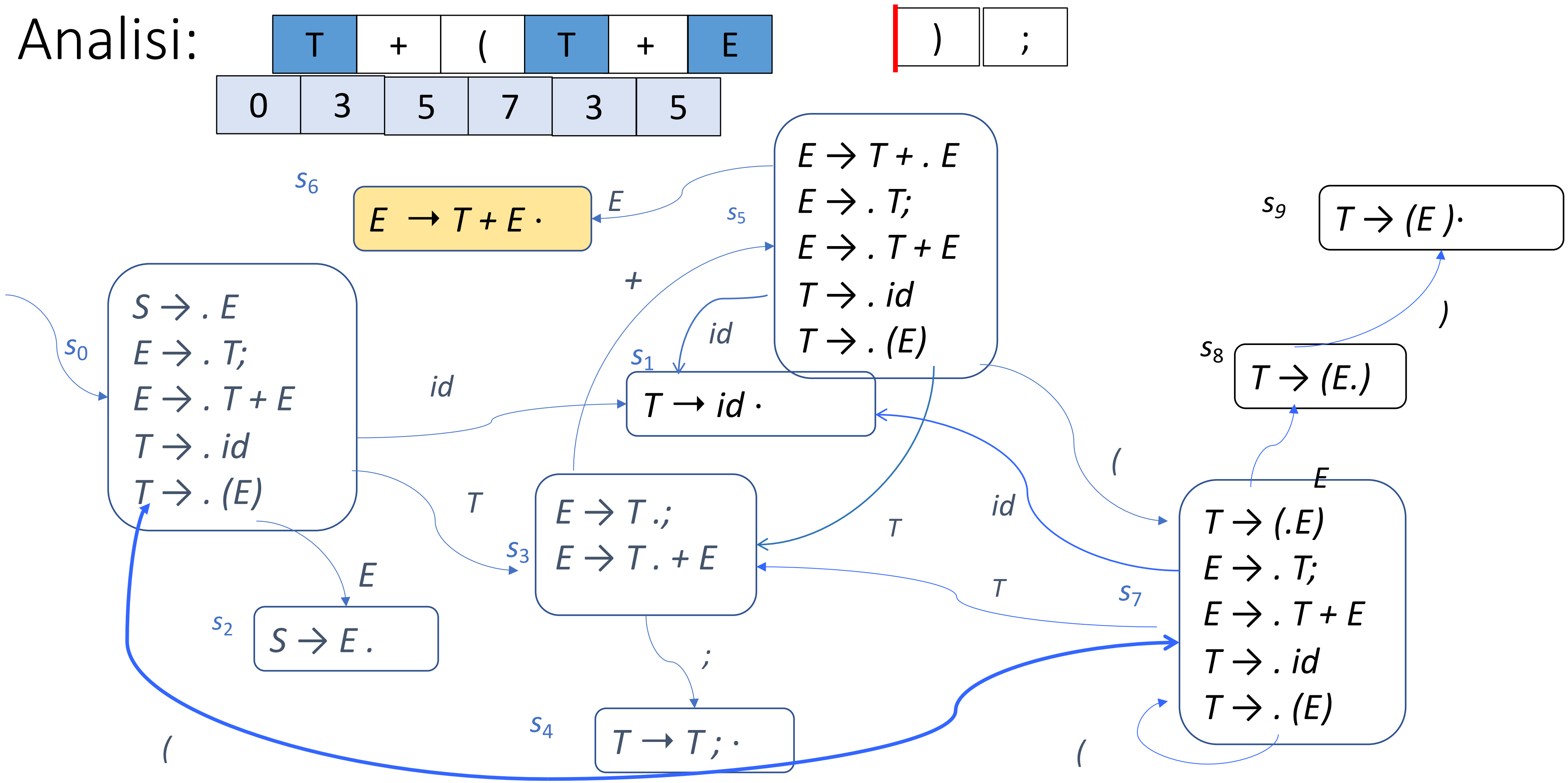
Analisi:

T	+	(	T	+	E
0	3	5	7	3	5

)	;



Analisi:

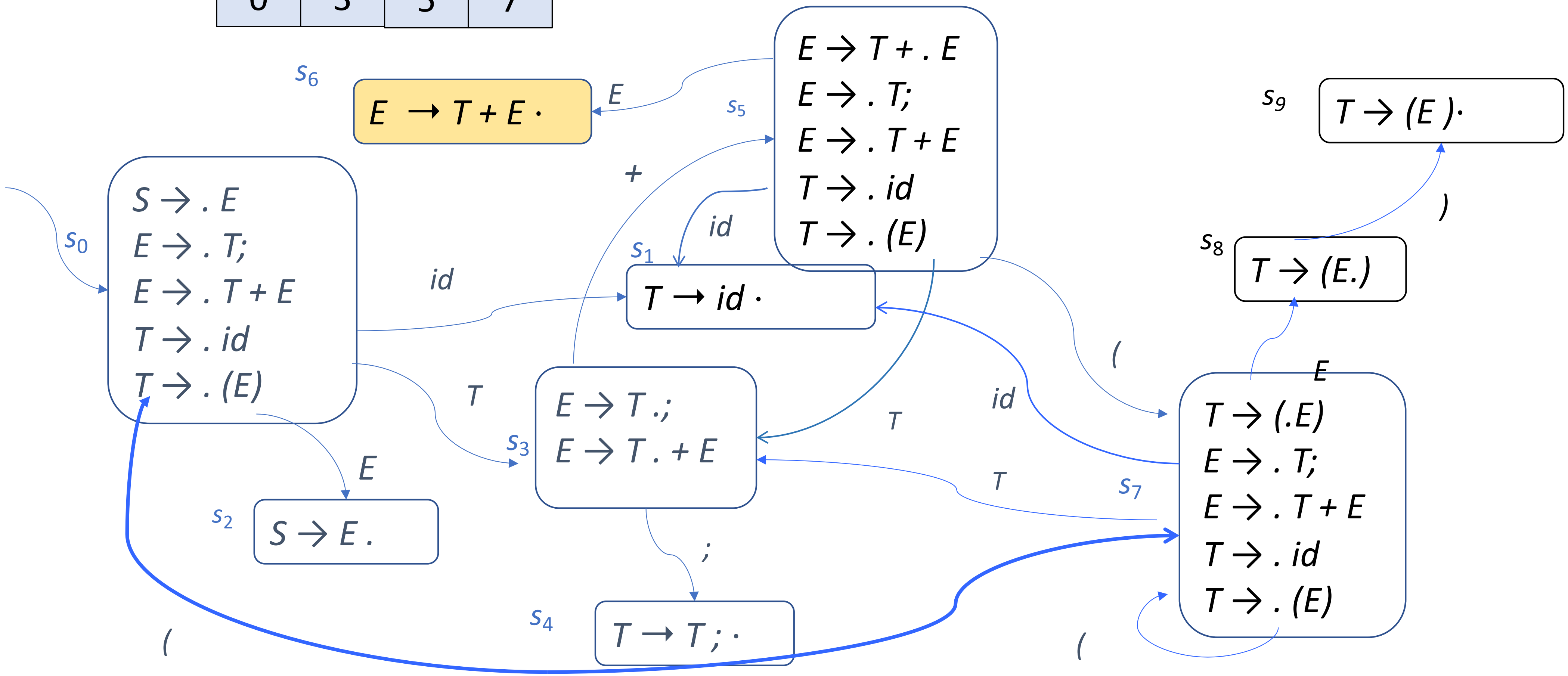

T | + | (| T | + | E
0 | 3 | 5 | 7 | 3 | 5
) | ;
s6
E → T + E ·
E
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s9
T → (E)·
+
)
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
id
id
s1
T → id ·
s8
T → (E.)
(
E
T
T
id
s3
E → T .;
E → T . + E
T
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
E
s2
S → E .
;
s4
T → T ; ·
(
(

Analisi:

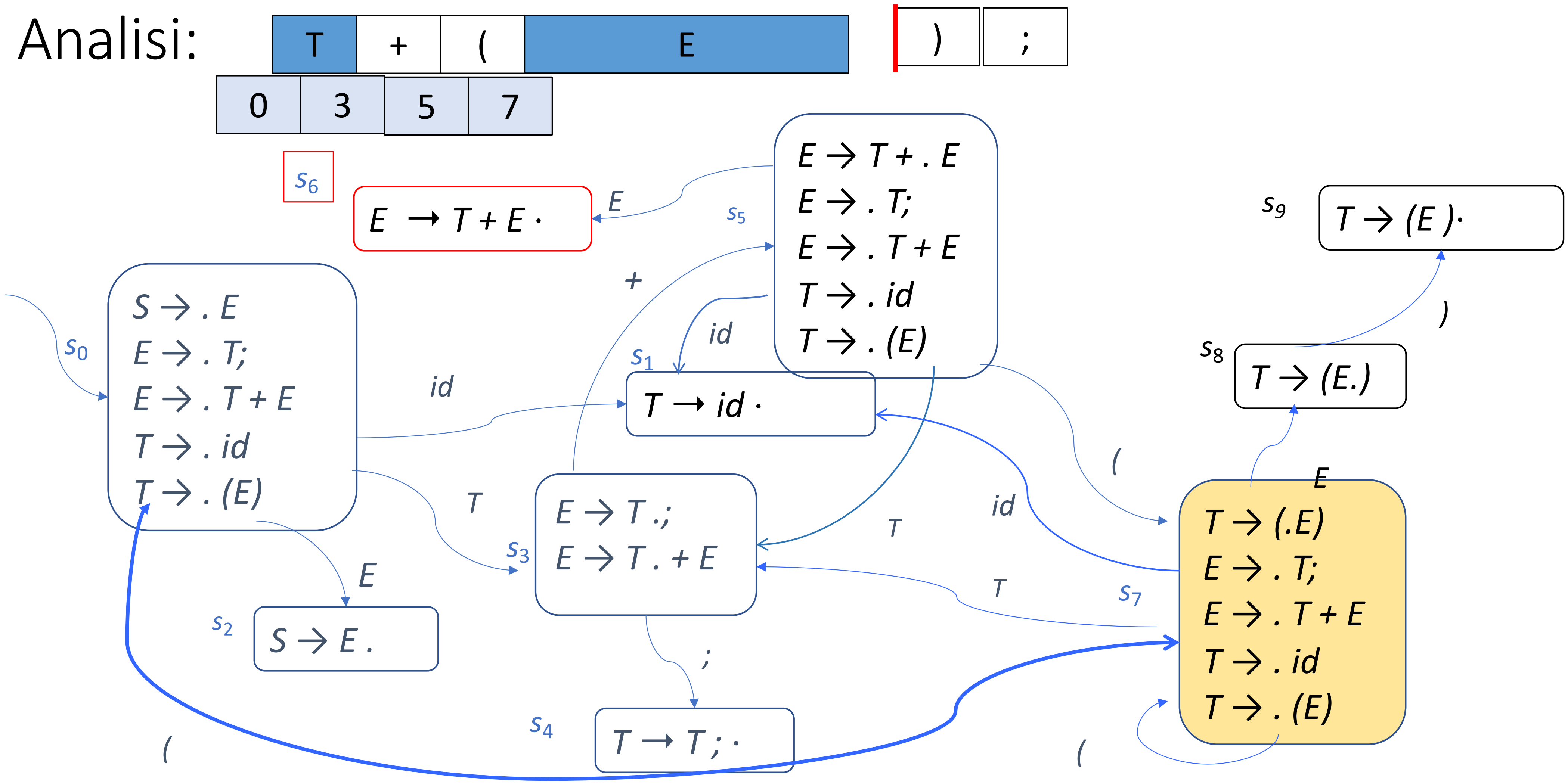
T	+	(

0	3	5	7

)	;

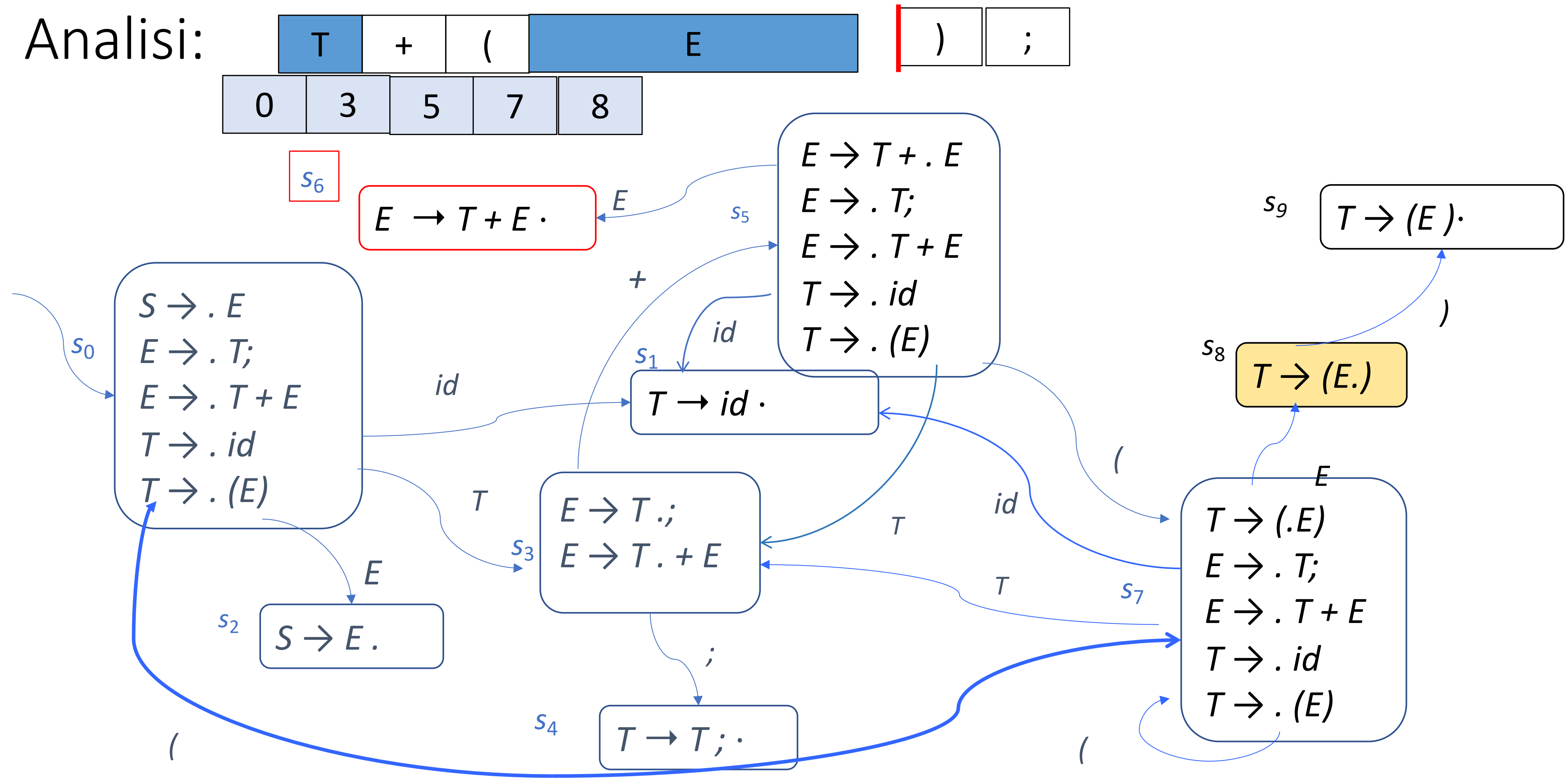


Analisi:



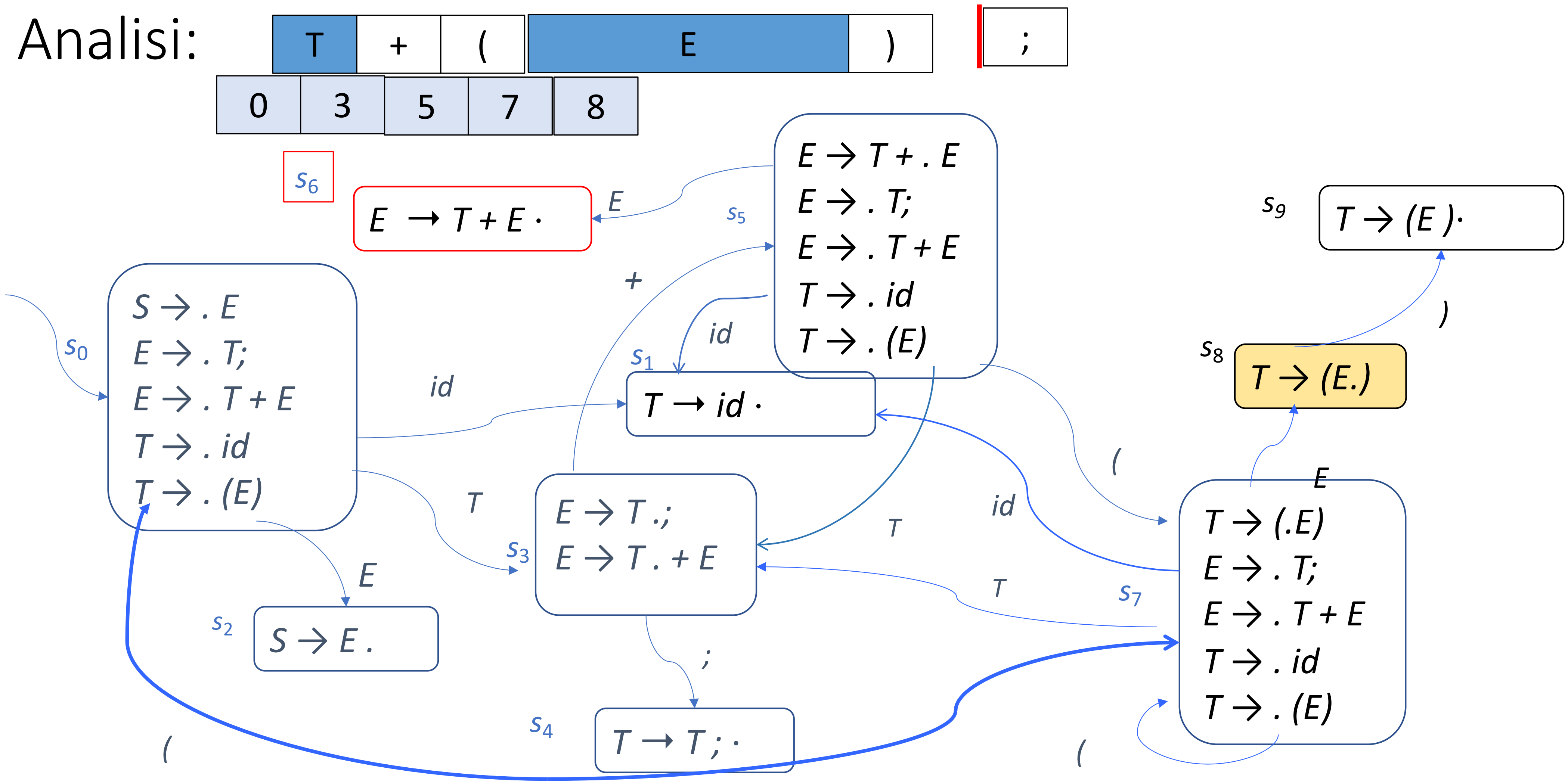
T
+
(
E
)
;
0
3
5
7
s6
E → T + E ·
E
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s9
T → (E)·
)
s8
T → (E.)
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
id
+
id
s1
T → id ·
(
T
s3
E → T .;
E → T . + E
T
id
E
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
T
E
s2
S → E .
;
s4
T → T ; ·
(
(

Analisi:



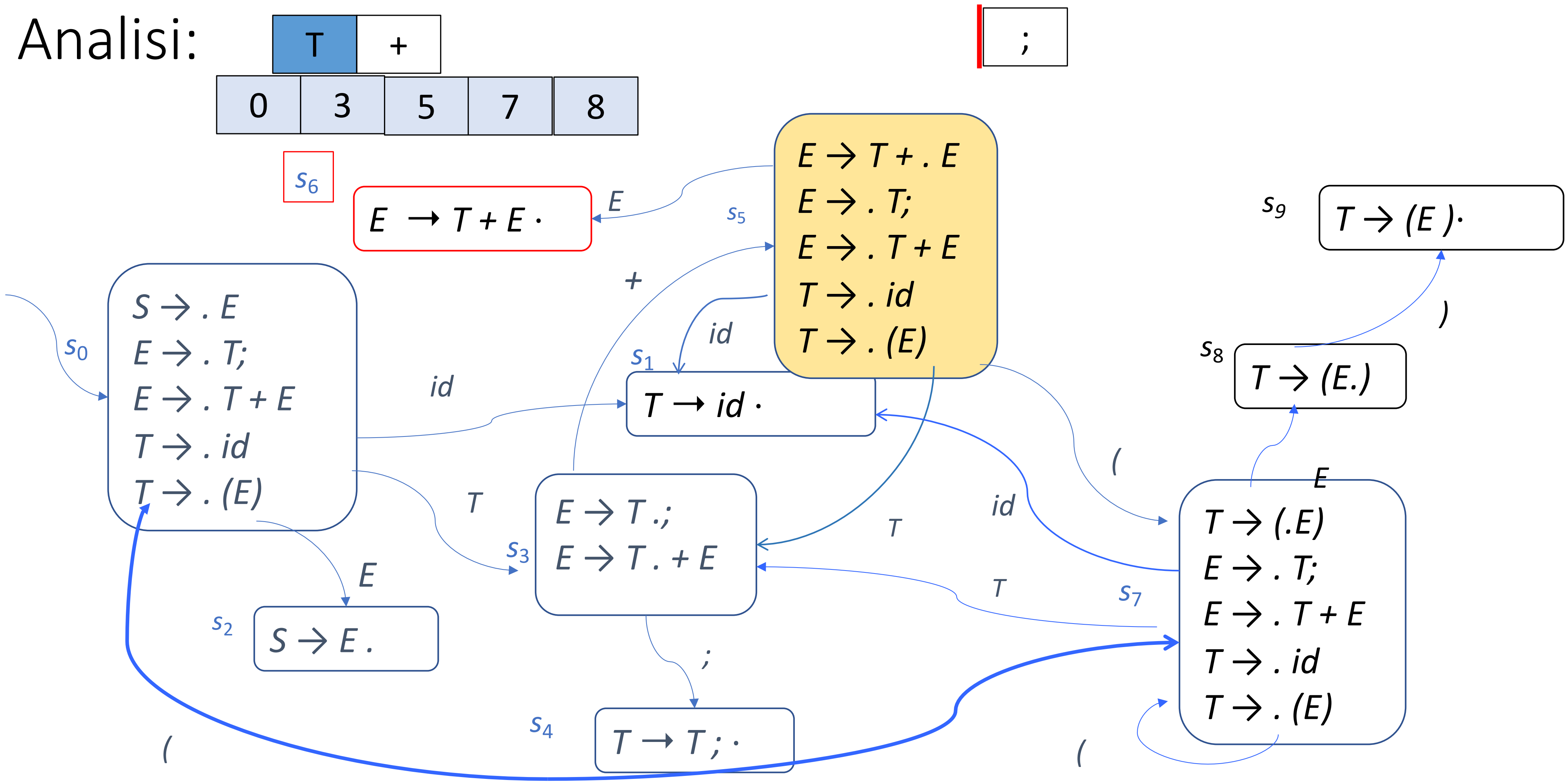
T
+
(
E
)
;
0
3
5
7
8
s6
E → T + E ·
E
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s9
T → (E)·
+
id
s1
T → id ·
s8
T → (E.)
)
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
id
(
E
T
s3
E → T .;
E → T . + E
T
id
T
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
E
s2
S → E .
;
s4
T → T ; ·
(
(

Analisi:

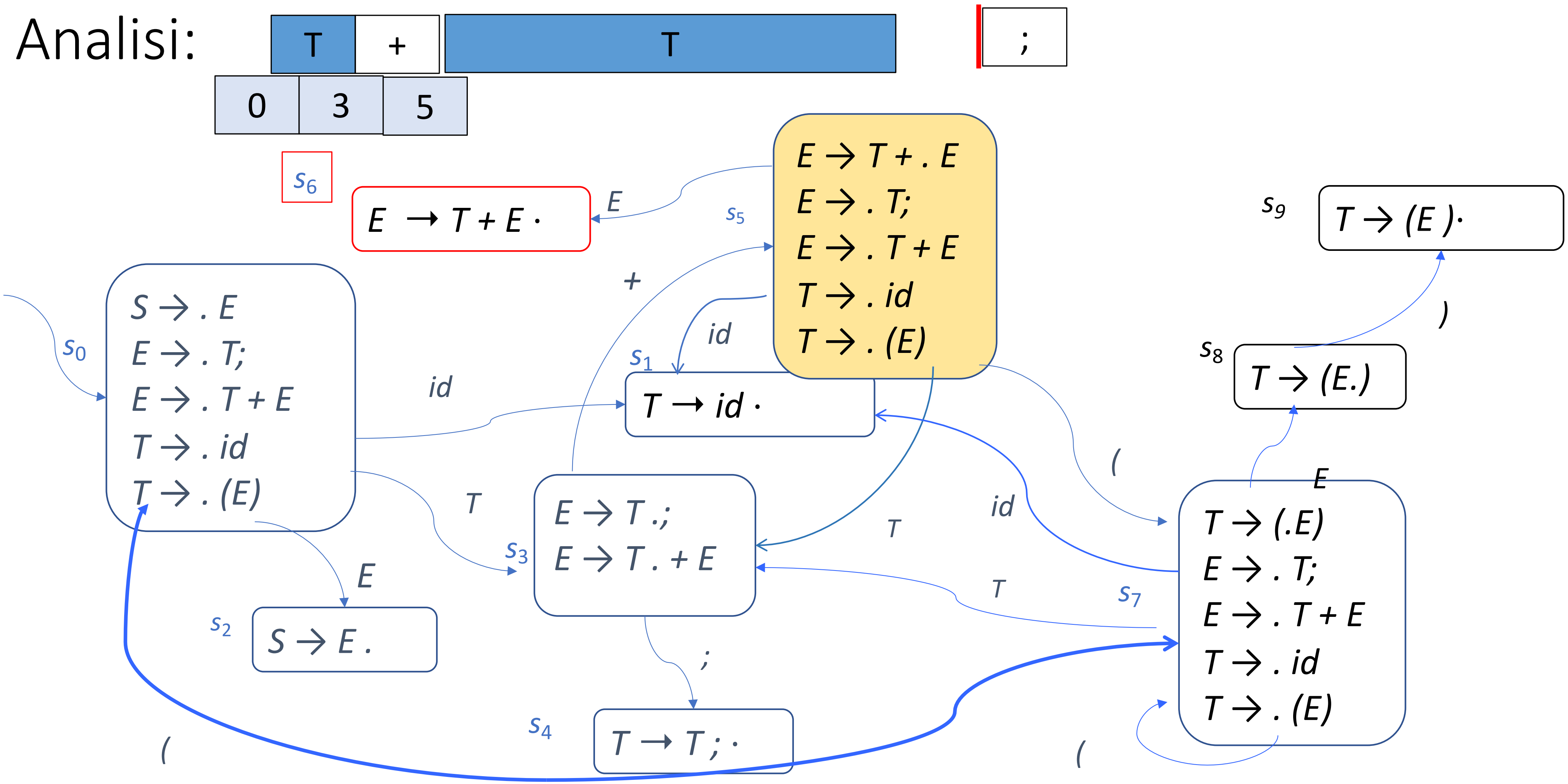


T
+
(
E
)
;
0
3
5
7
8
s6
E → T + E ·
E
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
+
id
s1
T → id ·
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
id
T
s3
E → T .;
E → T . + E
E
s2
S → E .
;
s4
T → T ; ·
(
T
id
T
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
(
(
E
s8
T → (E.)
)
s9
T → (E)·

Analisi:

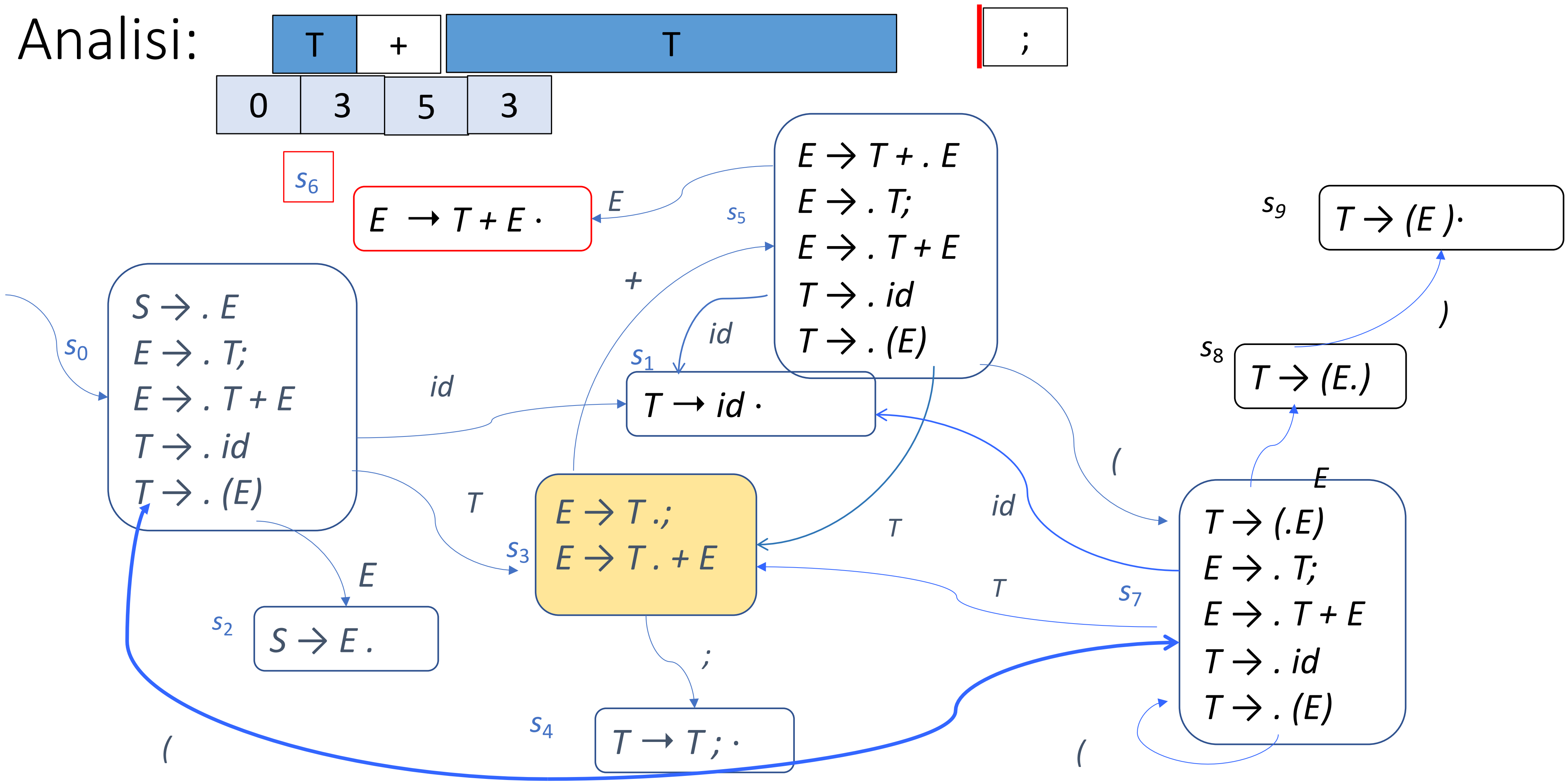

T
+
0
3
5
7
8
;
s_6
$E \rightarrow T + E \cdot$
E
s_5
$E \rightarrow T + . E$
$E \rightarrow . T;$
$E \rightarrow . T + E$
$T \rightarrow . id$
$T \rightarrow . (E)$
s_9
$T \rightarrow (E) \cdot$
+
id
)
s_8
$T \rightarrow (E.)$
s_0
$S \rightarrow . E$
$E \rightarrow . T;$
$E \rightarrow . T + E$
$T \rightarrow . id$
$T \rightarrow . (E)$
id
s_1
$T \rightarrow id \cdot$
(
E
T
s_3
$E \rightarrow T .;$
$E \rightarrow T . + E$
T
id
s_7
$T \rightarrow (.E)$
$E \rightarrow . T;$
$E \rightarrow . T + E$
$T \rightarrow . id$
$T \rightarrow . (E)$
T
E
s_2
$S \rightarrow E .$
;
s_4
$T \rightarrow T ; \cdot$
(
(

Analisi:



T
+
T
;
0
3
5
s_6
$E \to T + E \cdot$
E
s_5
$E \to T + . E$
$E \to . T;$
$E \to . T + E$
$T \to . id$
$T \to . (E)$
s_9
$T \to (E) \cdot$
)
s_8
$T \to (E.)$
+
id
s_1
$T \to id \cdot$
s_0
$S \to . E$
$E \to . T;$
$E \to . T + E$
$T \to . id$
$T \to . (E)$
id
(
E
T
s_3
$E \to T .;$
$E \to T . + E$
T
id
T
s_7
$T \to (.E)$
$E \to . T;$
$E \to . T + E$
$T \to . id$
$T \to . (E)$
E
s_2
$S \to E .$
;
s_4
$T \to T ; \cdot$
(
(

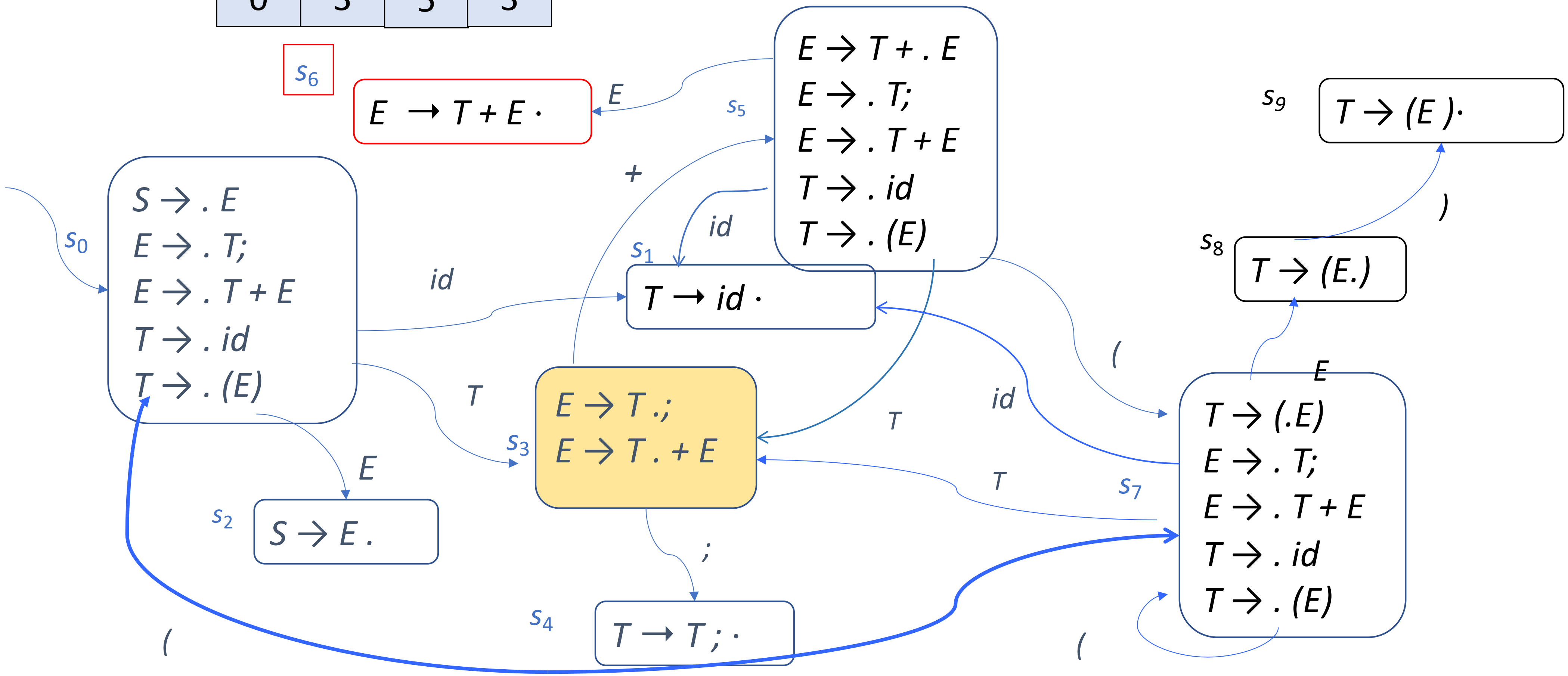
Analisi:



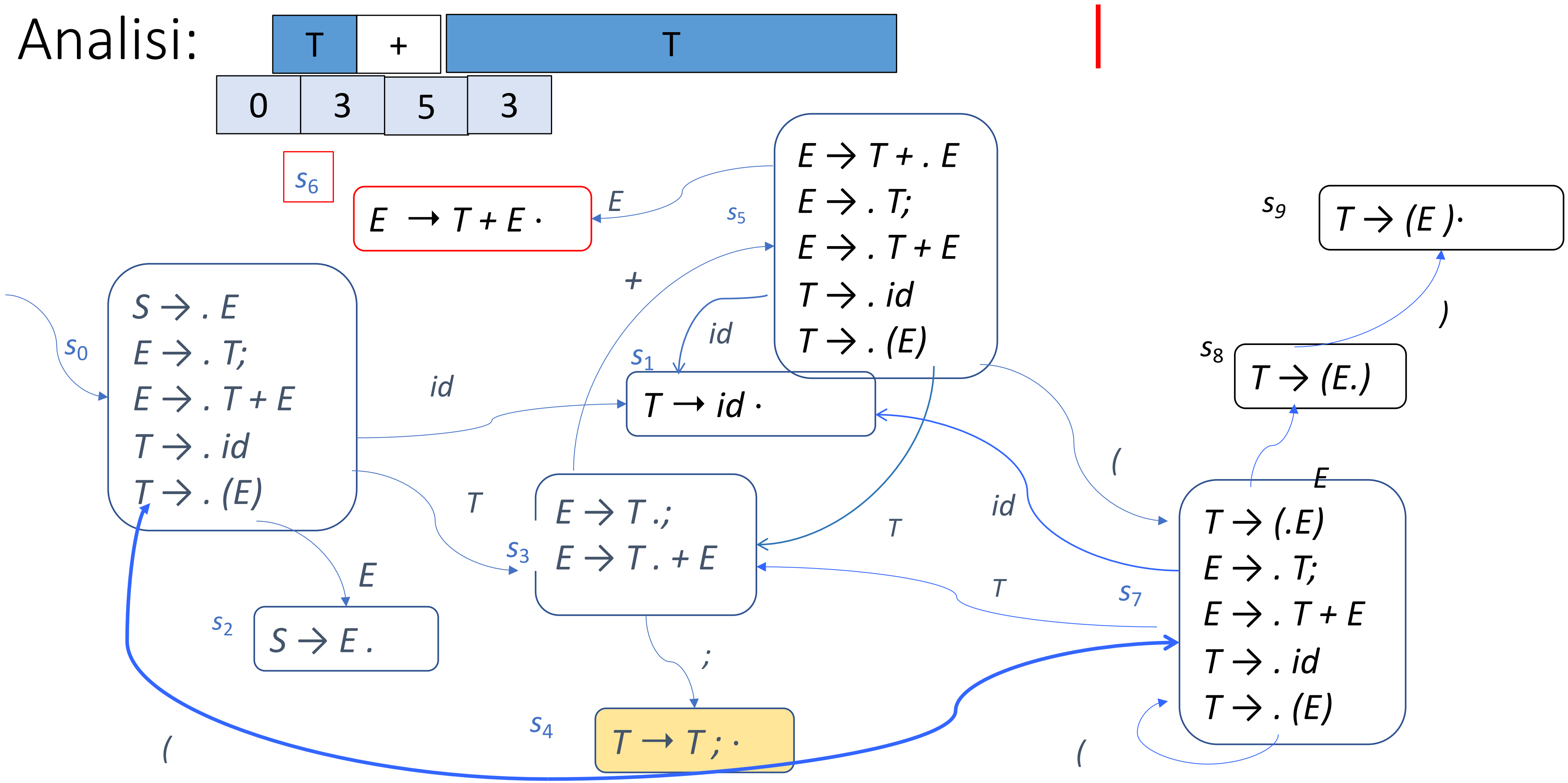
T
+
T
;
0
3
5
3
s_6
$E \rightarrow T + E \cdot$
s_5
$E \rightarrow T + . E$
$E \rightarrow . T;$
$E \rightarrow . T + E$
$T \rightarrow . id$
$T \rightarrow . (E)$
s_9
$T \rightarrow (E)\cdot$
s_8
$T \rightarrow (E.)$
s_0
$S \rightarrow . E$
$E \rightarrow . T;$
$E \rightarrow . T + E$
$T \rightarrow . id$
$T \rightarrow . (E)$
s_1
$T \rightarrow id \cdot$
s_3
$E \rightarrow T .;$
$E \rightarrow T . + E$
s_7
$T \rightarrow (.E)$
$E \rightarrow . T;$
$E \rightarrow . T + E$
$T \rightarrow . id$
$T \rightarrow . (E)$
s_2
$S \rightarrow E .$
s_4
$T \rightarrow T ; \cdot$
E
+
id
id
T
E
T
T
id
(
)
E
;
(
(

Analisi:

T	+	T	;
0	3	5	3



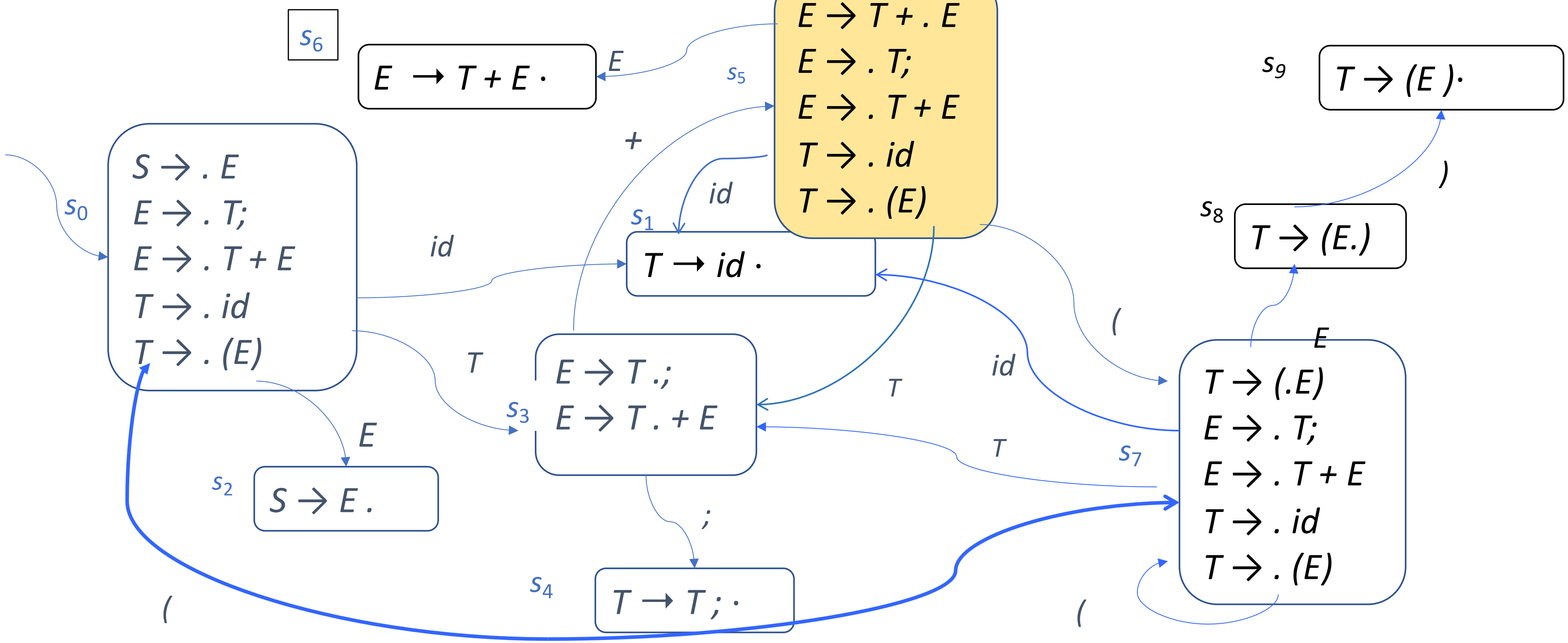
Analisi:



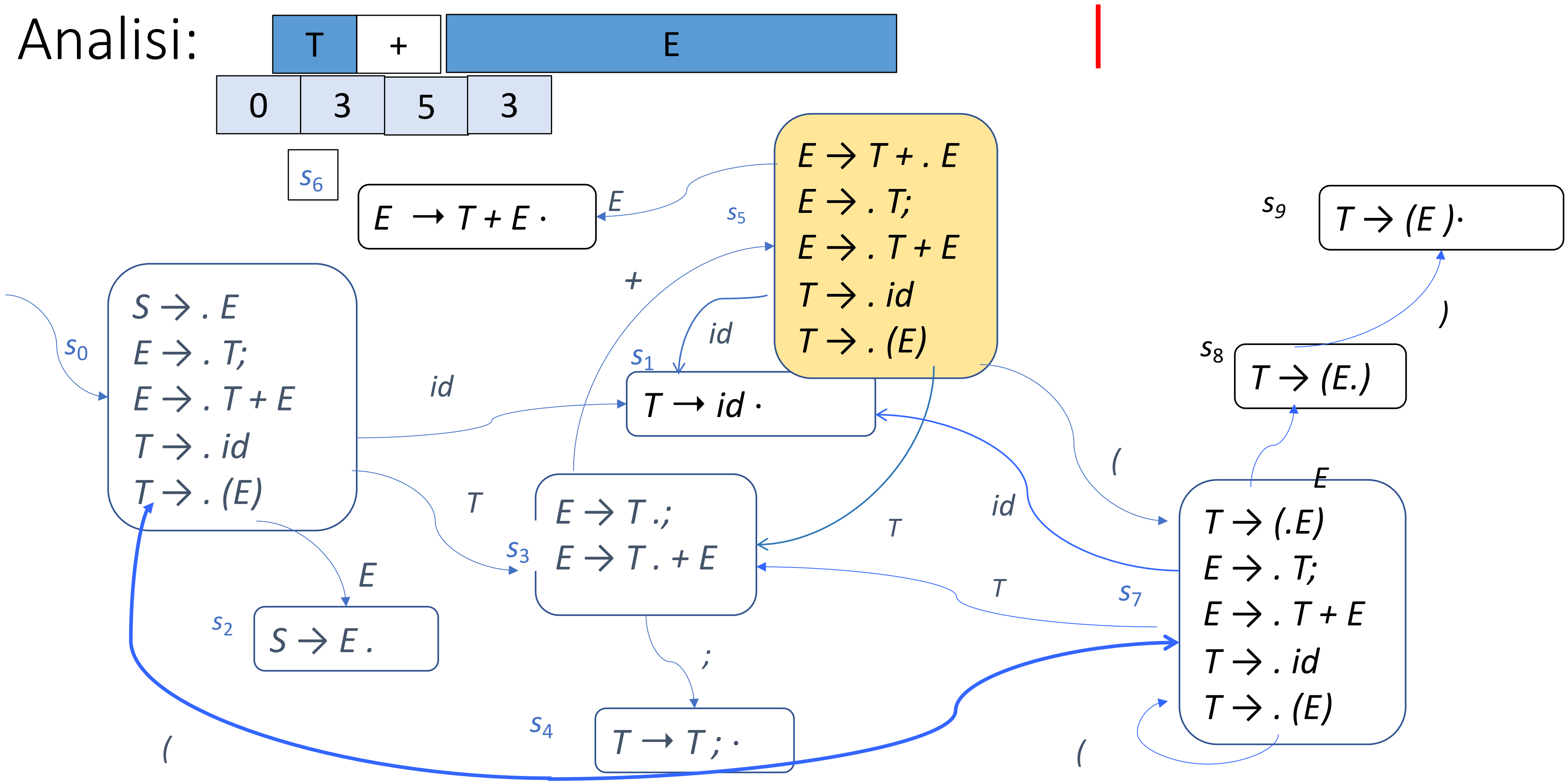
T
+
T
0
3
5
3
s_6
$E \to T + E \cdot$
E
s_5
$E \to T + . E$
$E \to . T;$
$E \to . T + E$
$T \to . id$
$T \to . (E)$
s_9
$T \to (E) \cdot$
)
s_8
$T \to (E.)$
s_0
$S \to . E$
$E \to . T;$
$E \to . T + E$
$T \to . id$
$T \to . (E)$
+
id
id
s_1
$T \to id \cdot$
(
E
T
s_3
$E \to T .;$
$E \to T . + E$
T
id
T
s_7
$T \to (.E)$
$E \to . T;$
$E \to . T + E$
$T \to . id$
$T \to . (E)$
E
s_2
$S \to E .$
;
s_4
$T \to T ; \cdot$
(
(

Analisi:

T	+		
0	3	5	3

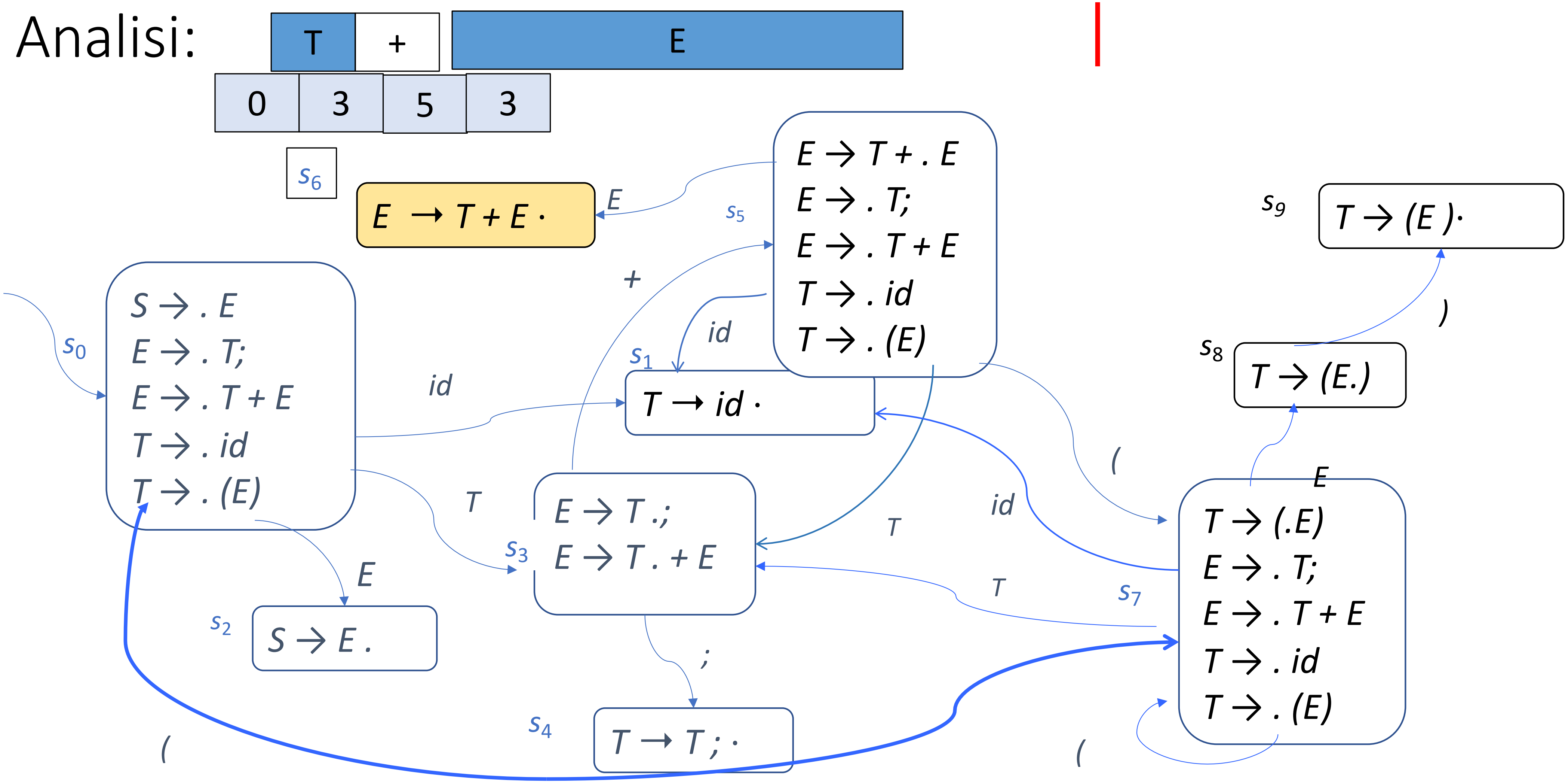


Analisi:



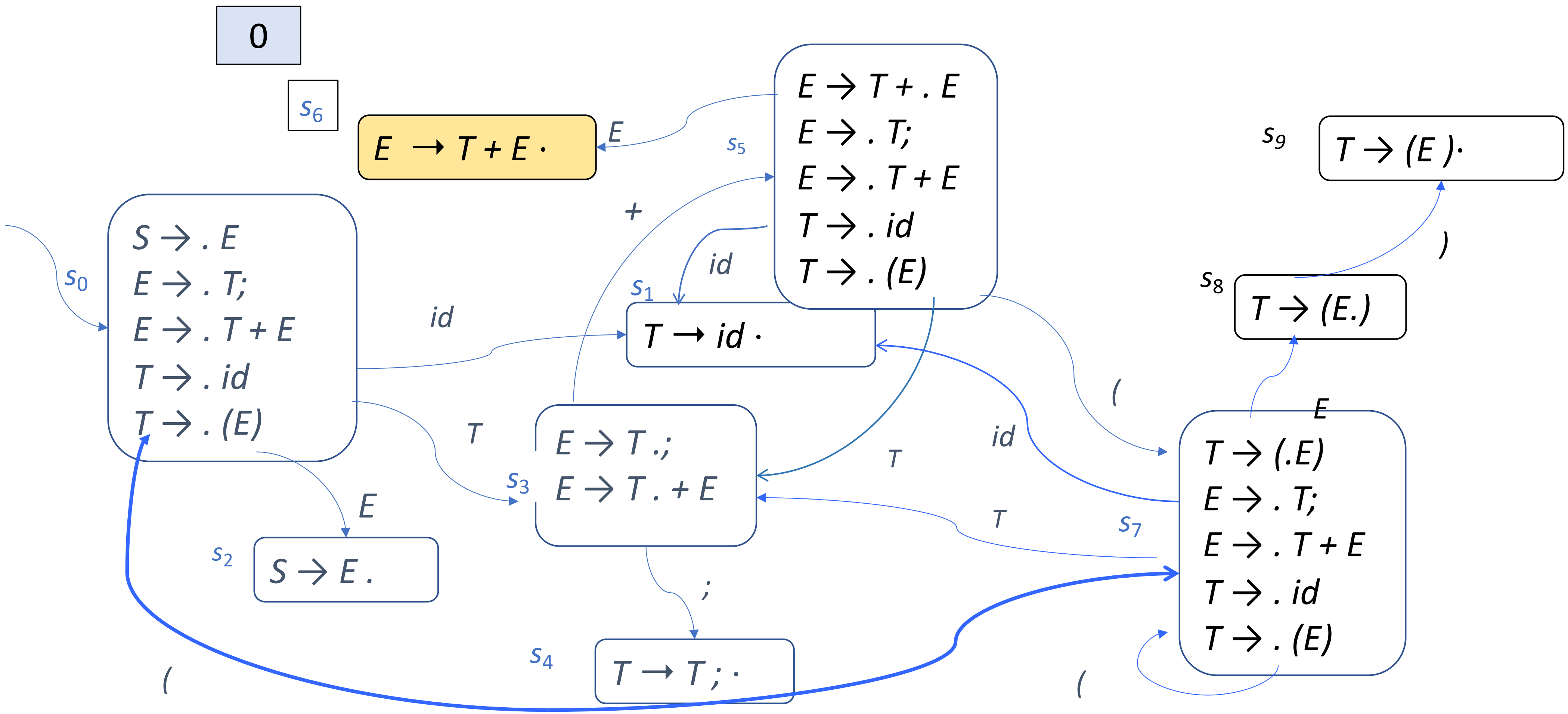
T
+
E
0
3
5
3
s6
E → T + E ·
E
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s9
T → (E)·
)
s8
T → (E.)
+
id
s1
T → id ·
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
id
(
E
T
s3
E → T .;
E → T . + E
T
id
T
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
E
s2
S → E .
;
s4
T → T ; ·
(
(

Analisi:

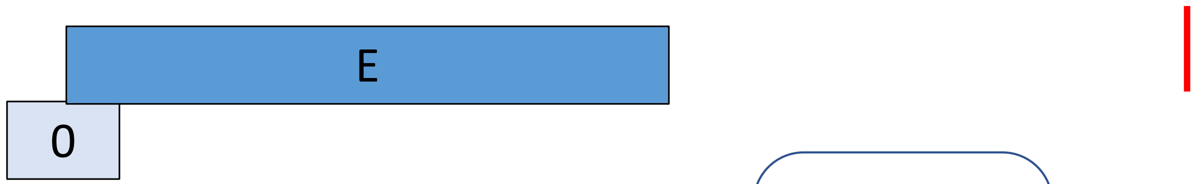


T
+
E
0
3
5
3
s6
E → T + E ·
E
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
+
id
s1
T → id ·
s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
id
T
s3
E → T .;
E → T . + E
E
s2
S → E .
;
s4
T → T ; ·
(
T
id
T
(
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
E
s8
T → (E.)
)
s9
T → (E)·
(

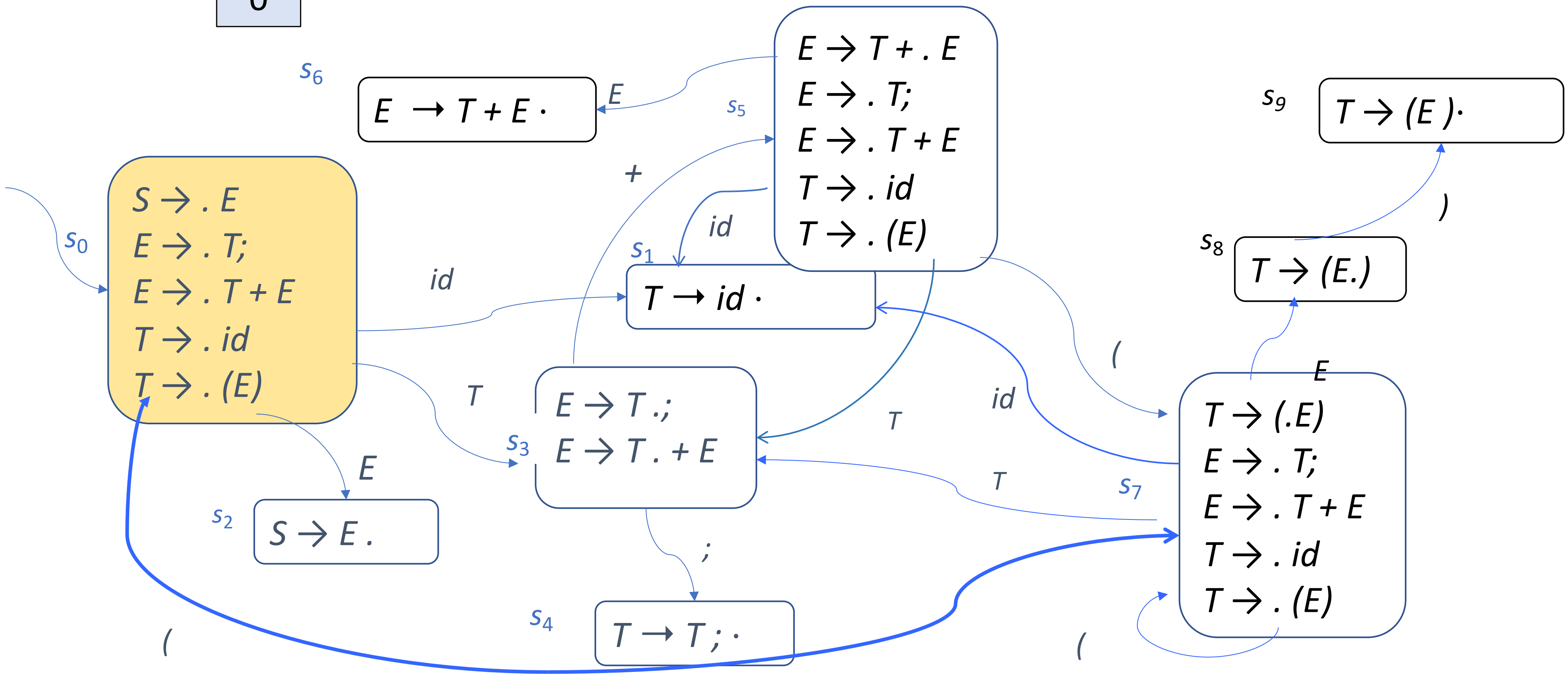
Analisi:


0
s_6
E → T + E ·
E
s_5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
s_9
T → (E)·
)
s_8
T → (E.)
s_0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
+
id
s_1
T → id ·
id
(
E
T
s_3
E → T .;
E → T . + E
T
id
T
s_7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
E
s_2
S → E .
;
s_4
T → T ; ·
(
(

Analisi:



E
0



s0
S → . E
E → . T;
E → . T + E
T → . id
T → . (E)
id
s1
T → id ·
T
s3
E → T .;
E → T . + E
E
s2
S → E .
;
s4
T → T ; ·
+
s5
E → T + . E
E → . T;
E → . T + E
T → . id
T → . (E)
E
s6
E → T + E ·
(
s7
T → (.E)
E → . T;
E → . T + E
T → . id
T → . (E)
s8
T → (E.)
)
s9
T → (E)·

Analisi:

- Siamo in stato 0
- Abbiamo completato analisi di S→E

L'analisi termina con successo!

Tavole Action e Goto

La tavola Action mappa ogni stato ad un'azione

- shift, esamina prossimo simbolo terminale
- reduce $A \to \alpha$, che esegue la riduzione $A \to \alpha$
- solo gli stati della forma $A \to \alpha$.
- tutti gli altri shift that reduction;

everything else shifts.

La tavola Goto mappa le coppie simbolo/stato dando il uovo stato

- La tavola è la tabella di transizione dell'automa

ESERCIZIO

Costruire le tavole per

il diagramma di transizione

precedente (automa)

grammatica LR(0)

- Una grammatica è detta LR(0) se, per ciascuno degli stati:
  - è presente al più una sola produzione con il punto finale (assenza di conflitti reduce-reduce)
  - non sono contemporaneamente presenti una produzione con il punto finale e un'altra con il punto non finale (assenza di conflitti shift-reduce)

- in una grammatica LR(0) la decisione shift/reduce viene presa senza guardare l'input (lookahead nullo)

Algoritmo di analisi LR(0) con tavole Action e Goto

Inizializza la pila con stato 0 e poni \$ alla fine dell'input

while (Action <> Accetta) and (Action <> Error) do

Sia la pila = $s_0 x_1 s_1 \ldots x_m s_m$ *e input rimanente =* $a_i a_{i+1} \ldots \$$

{$s_0\ s_1\ s_{2\ldots}$ *sono gli stati ;* $x_1\ x_2\ \ x_3$ *… sono le sequenze di terminali e non terminali}*

Considera la Tavola Action[s_m*,*a_i*]*

se Action[s_m *,* a_i*] è shift t : {Esegui shift di input e poni stato t in pila}*

se Action[s_m *,* a_i*] è reduce A → β: {esegui produzione A → β: pop |β| simboli da pila; sia t lo stato in cima alla pila dopo pop ; inserisci GOTO[t,A] in pila}*

se Action[s_m*,* a_i*] è accetta: accetta la stringa e termina*

se Action[s_m*,*a_i*] è non specificato (blank): segnala errore e termina*

end while

Limiti analisi LR(0)

- Se la grammatica non è LR(0) ci potrebbero essere più valori nella tavola Action. I possibili casi sono
- Sia shift che reduce sono presenti: conflitto *shift-reduce*. In questo caso l'analizzatore non sa decidere se continuare a eseguire shift o applicare una operazione di reduce (vedi caso espressioni aritmetiche considerato in precedenza)
- Analogamente un conflitto *reduce-reduce* si verifica quando l'analizzatore può scegliere fra più produzioni da applicare; questo caso si verifica tipicamente con le grammatiche ambigue (ma non solo)
- Un modo di risolvere i conflitti è riscrivere la grammatica oppure quello di analizzare l'input con maggiore dettaglio (andare avanti nela scansione dell'input).

LR(0) vs LR(k)

- il parsing LR(0), che usa lookahead nullo, si applica a poche grammatiche
  - in particolare: non funziona per la grammatica G' di slide 5, che è ambigua
- in generale, guardando $k \geq 1$ caratteri in input è possibile effettuare LR parsing su molte più grammatiche
  - NOTA Una grammatica ambigua non può naturalmente ammettere alcun tipo di parser, proprio a causa dell'ambiguità

alcuni enunciati (senza prova)

- *un linguaggio può essere generato da una grammatica LR(k) se e solo se è context free deterministico (riconosciuto da un automa a pila deterministico)*
- *un linguaggio ammette una grammatica LR(1) se e solo se ammette una grammatica LR(k), $k \geq 1$*

conseguenza:

- *tutti i linguaggi context free deterministici ammettono una grammatica generatrice LR(1)*

Grammatica per generare liste

La seguente grammatica genera liste annidate con sole x (nota grammatica aumentata); es. (x,x,x)$ (x,(x))$ (x,(x),((x),x))$

0: $S' \rightarrow S\$$ 1: $S \rightarrow (L)$ 2: $S \rightarrow x$ 3: $L \rightarrow S$ 4: $L \rightarrow L , S$

Terminali: x , ( ) $ (fine stringa)

1. Trovare derivazione sinistra di (x,(x))$
2. Date tabelle Action e Goto analizzare (x,(x))$
3. Costruire tabelle Action e Goto

Grammatica per generare liste

0: S’ → S$ 1: S → (L)
2: S → x 3: L → S
4: L → L , S

Date tabelle Action e Goto analizzare (x,(x))$

	(	)	x	,	$	S	L
0	s2		s1			goto3	
1	r2	r2	r2	r2	r2		
2	s2		s1			goto6	goto4
3					Accetta		
4		s5		s7			
5	r1	r1	r1	r1	r1		
6	r3	r3	r3	r3	r3		
7	s2		s1			goto8	
8	r4	r4	r4	r4	r4		

Stato	Action	Goto

pila	input	action
0	(x,(x))$	s2

In tabella riga 0 colonna (c’è s2
s2= shift input. Quindi metti in pila (- e vai a stato 2
Otteniamo in pila 0(2

0(2	x,(x))$	s1

Infatti riga 2, colonna x c’è s1
(analogo a prima)

Grammatica per generare liste

0: S' → S$ 1: S → (L)
2: S → x   3: L → S
4: L → L , S

Date tabelle Action e Goto analizzare (x,(x))$

	(	)	x	,	$	S	L
0	s2		s1			goto3	
1	r2	r2	r2	r2	r2		
2	s2		s1			goto6	goto4
3					Accetta		
4		s5		s7			
5	r1	r1	r1	r1	r1		
6	r3	r3	r3	r3	r3		
7	s2		s1			goto8	
8	r4	r4	r4	r4	r4		

Stato	Action	Goto

pila	input	action
0	(x,(x))$	s2
0(2	x,(x))$	s1
0(2x1	,(x))$	r2: S→x

Infatti riga 1 col. , c'è r2 - reduce produzione 2: S→x. Quindi

- togli x1 dalla pila e in pila trovi 2
- In tabella riga 2 e S (lato sinistro di prod.2) trovi goto6; quindi metti in pila S e 6 e ottieni)(2S6

0(2S6	,(x))$	r3: L→S

(analogo)

Grammatica per generare liste

0: S' → S\$ 1: S → (L)
2: S →x 3: L →S 4: L → L , S

2.Date tabelle Action e Goto analizzare (x,(x))\$

	(	)	x	,	\$	S	L
0	s2		s1			goto3	
1	r2	r2	r2	r2	r2		
2	s2		s1			goto6	goto4
3					Accetta		
4		s5		s7			
5	r1	r1	r1	r1	r1		
6	r3	r3	r3	r3	r3		
7	s2		s1			goto8	
8	r4	r4	r4	r4	r4		

Stato	Action	Goto

pila	input	action
0	(x,(x))\$	s2
0(2	x,(x))\$	s1
0(2x1	,(x))\$	r2: S→ x
0(2S6	,(x))\$	r3: L→ S
0(2L4	,(x))\$	s7
0(2L4,7	(x))\$	s2
0(2L4,7(2	x))\$	s1
0(2L4,7(2x1	))\$	r2: S → x
0(2L4,7(2S6	))\$	r3: L→ S
0(2L4,7(2L4	))\$	s5
0(2L4,7(2L4)5	)\$	r1: S → (L)
0(2L4,7S8	)\$	r4: L→ L,S
0(2L4	)\$	s5
0(2L4)5	\$	r1:S→ (L)
03S	\$	Accetta

Algoritmo per generare le tabelle Action e Goto

Per generare le tavole Action e Goto due passi

1. costruzione diagramma delle transizioni fra stati
2. costruzione tabelle di analisi (tabelle Action e Goto)

- *Gli stati rappresentano possibili avanzamenti nell'analisi di una produzione (si usa il punto per segnalare a che punto siamo con l'analisi)*
- *Gli shift e i goto esplicitamente connettono gli stati; arco **shift** è etichettato con simbolo terminale / arco **reduce** con simbolo non terminale*
- *Le operazioni di Reduce implicitamente muovono in un altro stato con operazioni pop sulla pila dopo cui si usa la tabella goto per produrre il nuovo stato*

Algoritmo per generare le tabelle Action e Goto

1. costruzione diagramma delle transizioni fra stati

Cosa è uno stato?: uno stato è qualcosa del tipo [$A \rightarrow \alpha.B\beta,a$] e rappresenta la previsione di eseguire la produzione $A \rightarrow \alpha B\beta$

- Il punto prima di B segnala che α è già stato esaminato ed è in pila
- Ci aspettiamo di vedere simboli che sono genarati da B
- Se $B\beta$ è ε (vuoto) e se il prossimo simbolo è un non terminale applica la produzione e riduci A

Costruzione dell'insieme di stati:

1. Inizia con lo stato iniziale: prendi produzione da assioma e considera la sua chiusura
2. Determina il prossimo stato a partire da stato iniziale esaminando le possibili produzioni; esegui la chiusura e determina un nuovo stato
3. Itera il passo precedente fino a quando tutte le transizioni sono state esaminate

Algoritmo per generare le tabelle Action e Goto

2. costruzione tabelle di analisi (tabelle Action e Goto)

- Gli shift e i goto esplicitamente connettono gli stati ; shift corrisponde ad arco con simbolo terminale / reduce ad arco con simbolo non terminale
- Ricorda se Action[s_m,a_i] = reduce A → β .

  allora si esegue produzione A → β: pop |β| simboli da pila; se t è lo stato in cima alla pila dopo pop si inserisce GOTO[t,A] in pila}

  per ogni arco X(I, J) del diagramma ottenuto

  se X è terminale, metti shift J a (I, X)
  se X è non-terminale, metti goto J a (I, X)
  se I contiene S'→ . $, metti accetta a (I, $)
  se I contiene A → α . dove A→ α . è la produzione n della grammatica (analisi lato destro completata infatti il . è alla fine)

  per ogni terminale x, metti reduce n a (I, x)

0: S’ → S$ 1: S → (L)
2: S → x 3: L → S
4: L → L , S

Grammatica per generare liste

3. Costruire tabelle Action e Goto

Consideriamo la grammatica in alto a sinistra -simboli terminali (,),x

- Si inizia da stato 0 con l’unica produzione dall’assioma S’ S’→S$
- Usiamo la notazione punto abbiamo: S’ → . S$ (le produzioni con “.” sono dette item e indicano cosa c’è nello stack - a sinistra del . e cosa ci aspettiamo in input – a destra del .
- L’input può iniziare con qualunque cosa con cui può iniziare S (ricorda insieme First). In questo caso abbiamo x o (
- Quindi effettuiamo la chiusura inserendo nello stato 0:

  S→.x$ e S→.(L)$
- Alla fine otteniamo lo stato 0 composto da

  S’→.S$ S→.x$ S→.(L)

Grammatica per generare liste

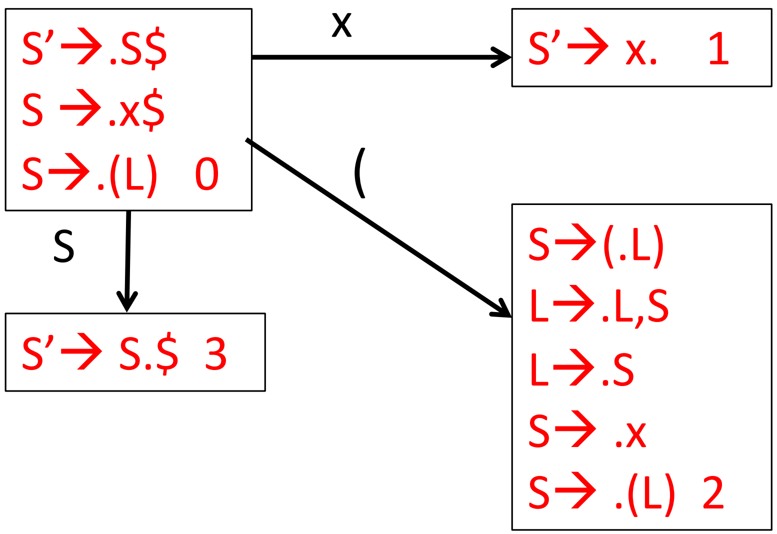
0: S' → S$ 1: S → (L)
2: S → x 3: L → S
4: L → L , S

3. Costruire tabelle Action e Goto

- Nello stato 0: S'→.S$ S'→.x$ S→.(L) possiamo avere come successivo carattere di input x o (oppure riduzione su produzione S'→S$
- Questo corrisponde a creare tre stati
- Stato 1: da stato 0 con input x a stato 1 S'→ x. (shift) - in questo stato siamo alla fine di un item (il . è in fondo) quindi non dobbiamo fare chiusura e dopo faremo reduce
- Stato 2: da stato 0 con input (andiamo in S→(.L) (shift) - effettuiamo la chiusura su L e otteniamo S→(.L) L→.L,S L→.S S→ .x S→ .(L)
- Stato 3: da stato 0 possiamo eseguire reduce su S - consideriamo la produzione S' → S$ della grammatica ottenendo S'→ S.$ - nota si utilizza notazione '.': in stato 0 ho S'→.S$ applicarla permette di ottenere S→S.$

Grammatica per generare liste

0: S’ → S$ 1: S → (L)
2: S →x 3: L →S
4: L → L , S



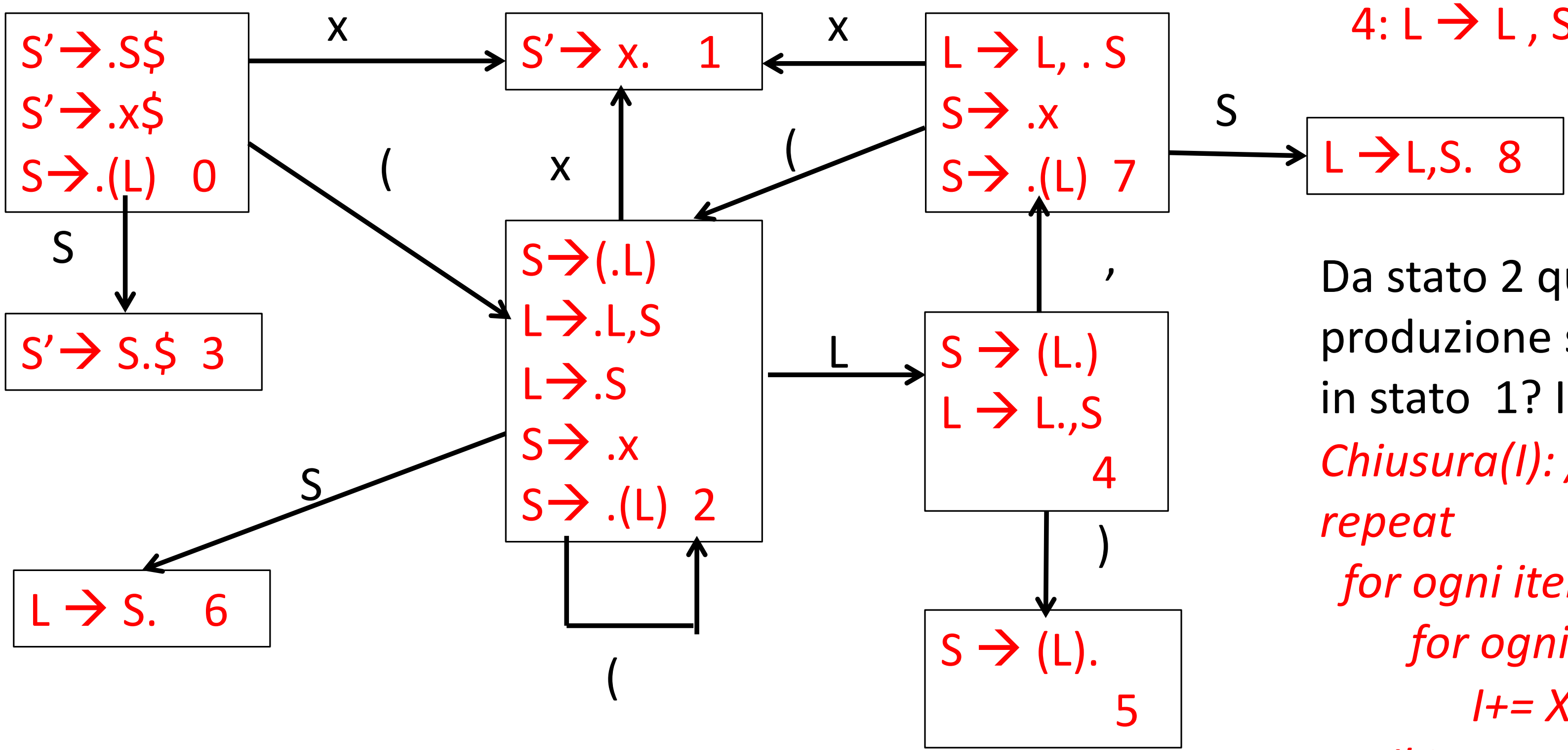
Come calcolo produzioni di stato 2?

- Parto da S→(.L) (ottenuto da stato 0)
- dopo il punto ho L simb. nonterminale presente nel lato sinistro di due prod. L→L,S e L→S; aggiungo L→.L,S e L→.S (con punto all’inizio del lato sinistro)
- L→.L,S ha L dopo il punto (già fatto)
- L→.S miichiede di aggiungere chiusura di S

- *Gli shift e i goto esplicitamente connettono gli stati ; shift arco con simbolo terminale / reduce arco con simbolo non terminale*
- *Reduce implicitamente muovono in un altro stato con operazioni pop sulla pila dopo cui si usa la tabella goto per produrre un nuovo stato*

Grammatica per generare liste

0: S’ → S$ 1: S → (L)
2: S → x 3: L → S
4: L → L , S



Da stato 2 quale (o quali) produzione si utilizza per andare in stato 1? In stato 4? Stato 2?

Chiusura(I): // stato I
repeat
 for ogni item A →α . Xβ
  for ogni X → γ
   I+= X → . γ
until I non cambia

Formule booleane CNF

Data la grammatica

$F' \rightarrow \cdot F\$$

$F \rightarrow \cdot F \wedge C \mid C$

$C \rightarrow \cdot(L \vee L \vee L)$

$L \rightarrow \cdot id \mid \cdot \neg id$

Nota grammatica

- con assioma presente in una sola produzione
- con simbolo $ alla fine della stringa in input

Formule booleane CNF

s_0

F' → ·F$
(+) F → ·F∧C | ·C
(+) C → ·(L∨ L ∨ L)

Data la grammatica

F' → ·F$

F → ·F∧C | C

C → ·(L ∨ L ∨ L)

L → ·id | ·¬id

Nota grammatica

- con assioma presente in una sola produzione
- con simbolo $ alla fine della stringa in input

Stato 0 include F' → ·F$

- dobbiamo aggiungere la sua chiusura
- la chiusura indica cosa può seguire nell'analisi dopo il punto in questo caso cercare cosa può seguire dopo F
- le produzioni coinvolte sono F → ·F∧C e F → ·C che sono inserite in stato 0
- Considerare F → ·C implica che dobbiamo anche inserire produzioni con C a sinistra di →

Formule booleane CNF

F' → F$
F → F∧C | C
C → (L ∨ L ∨ L)
L → id | ¬id

s_0

F' → ·F$
(+) F → ·F∧C | ·C
(+) C → ·(L∨ L ∨ L)

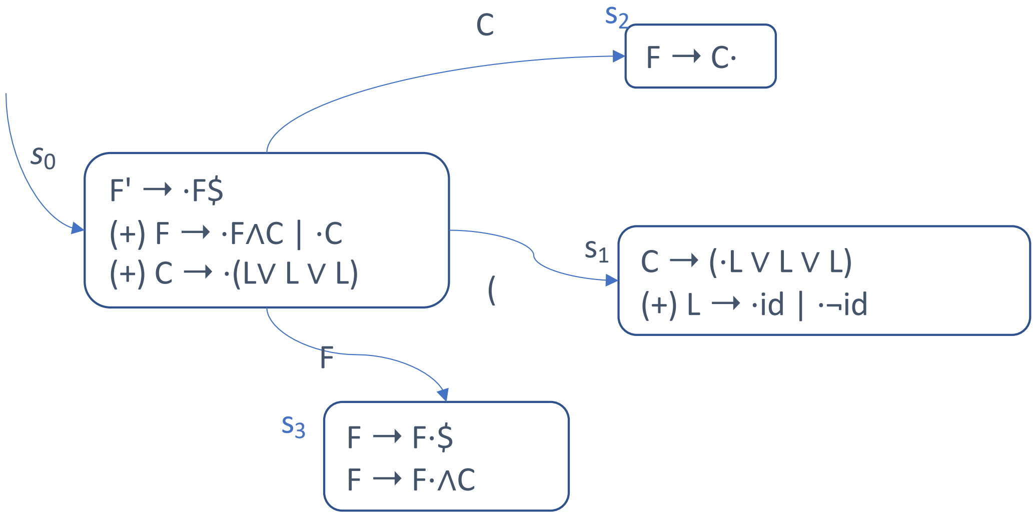
Stato 0: F' → . FS

- Bisogna aggiungere la chiusura di F e considerare le due produzioni
  1) F → ·F∧C e 2) F→C
- La chiusura di 1) non aggiunge altro
- La seconda richiede di aggiungere la chiusura di C e considerare la produzione C → ·(L ∨ L ∨ L)

Come proseguire? Quanti archi uscenti da s0? Basta guardare i simboli nella parte destra subilto dopo il punto e le produzioni relative (usando sempre notazione punto)
I simboli dopo il . In s0 sono : F, C e (
per F' le produzioni sono : F'→ F·∧C e F' → F·$ per C abbiamo : F→ C
Per (abbiamo C → ·(L∨ L ∨ L)

Formule booleane CNF

$F' \rightarrow \cdot F\$$
$F \rightarrow \cdot F \wedge C \mid C$
$C \rightarrow \cdot(L \vee L \vee L)$
$L \rightarrow \cdot id \mid \cdot\neg id$



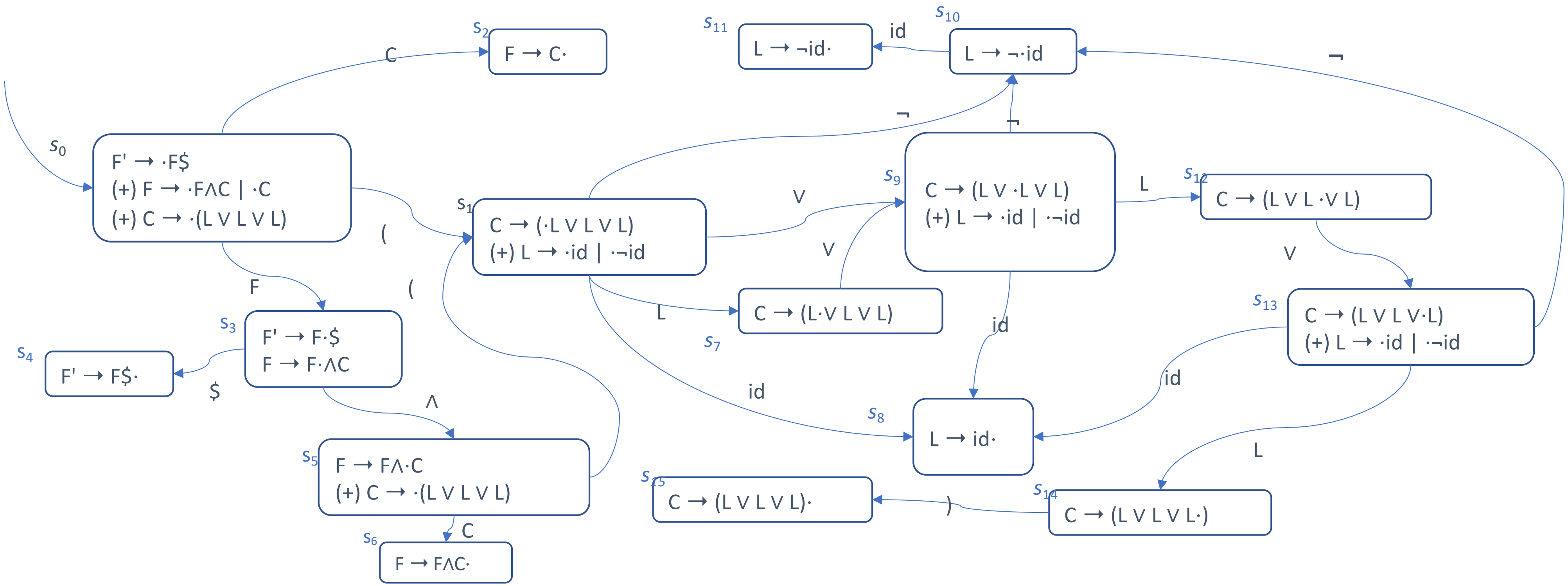
Stato 0: F’ → . FS

- Bisogna aggiungere la chiusura di F e considerare le due produzioni 1) $F \rightarrow \cdot F \wedge C$ e 2) $F \rightarrow C$
- La chiusura di 1) non aggiunge altro
- La seconda richiede di aggiungere la chiusura di C e considerare la produzone

  $C \rightarrow \cdot(L \vee L \vee L)$

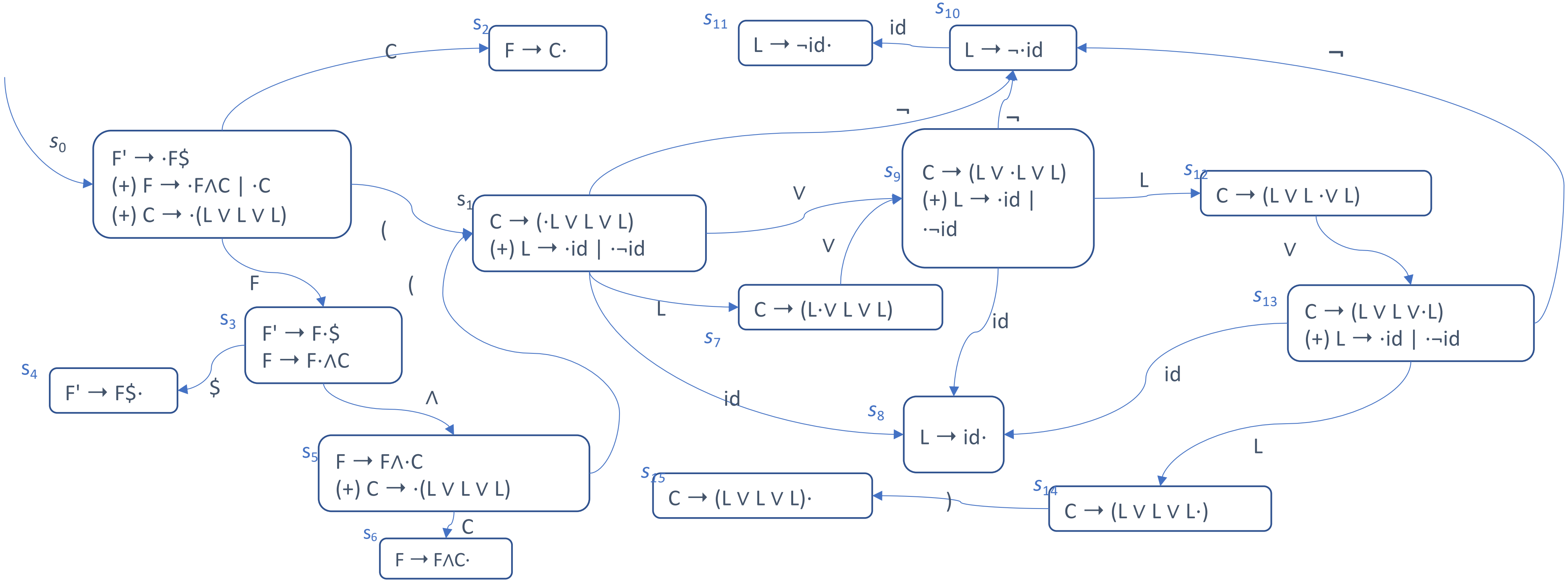
Formule booleane CNF

F' ⟶ ·F$
F ⟶ ·F∧C | C
C ⟶ ·(L ∨ L ∨ L)
L ⟶ ·id | ·¬id



Formule booleane CNF

F' ⟶ ·F$
F ⟶ ·F∧C | C
C ⟶ ·(L ∨ L ∨ L)
L ⟶ ·id | ·¬id



Grammatiche ambigue generano conflitti

Esempio

Considera la nostra grammatica amnbigua preferita

$E \rightarrow E+E \mid E*E \mid (E) \mid int$

Analisi di : int * int + int

Derivazio e 1:

| int * int + int ...	shift
.....	....
E * E | + int	reduce E → E * E
E | + int	shift
E + | int	shift
E + int |	reduce E → E
E + E |	reduce E→ E + E
E |	FINE

Grammatiche ambigue generano conflitti

Esempio

Considera la nostra grammatica amnbigua preferita

E→E+E | E*E |(E) | int

Analisi di : int * int + int

Derivazione 2:

| int * int + int ...	shift
.....	....
E * E | + int shift	
E * E + | int	shift
E * E + int |	reduce E→ int
E * E + E |	reduce E → E + E
E * E |	reduce E→ E * E
E |	FINE