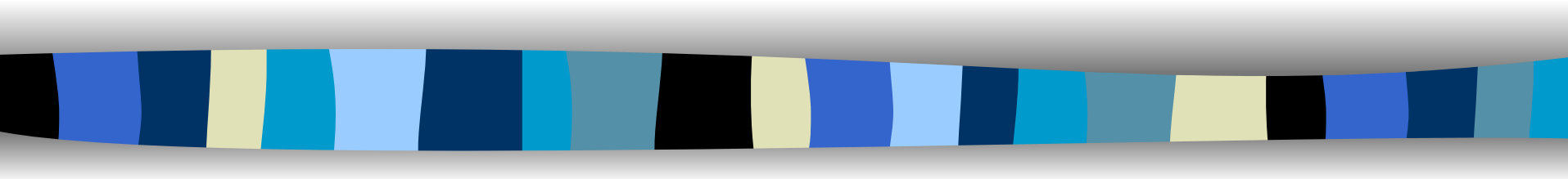
Bottom-up Parsing



Viable Prefixes, conflitti e formato delle tabelle per il parsing LR

Viable Prefixes

- Prima di continuare finiamo il discorso sul parsing shift-reduce in generale
- C'è una ragione importante per cui abbiamo usato uno stack per il parser shift-reduce:
- La maniglia da ridurre apparirà sempre sulla testa dello stack, mai all'interno

Viable Prefixes

- Consideriamo le possibili forme di due passi successivi in una derivazione rightmost:

1. $S \Rightarrow^*_{rm} \alpha Az \Rightarrow_{rm} \alpha\beta Byz \Rightarrow_{rm} \alpha\beta\gamma yz$

- A è dapprima riscritto con βBy e poi, essendo B il non terminale più a destra, viene riscritto con una certa γ

2. $S \Rightarrow^*_{rm} \alpha BxAz \Rightarrow_{rm} \alpha Bxyz \Rightarrow_{rm} \alpha\gamma xyz$

- A è riscritto in una stringa y di soli terminali e quindi al passo successivo verrà riscritto il primo simbolo non terminale che precedeva A, cioè B

Viable Prefixes

- Consideriamo il caso 1. al rovescio. Supponiamo che il parser ha raggiunto la seguente configurazione:
- STACK: $\$\alpha\beta\gamma$ yz$: INPUT
- Il parser deve ridurre la handle γ a B e raggiunge la configurazione:
- STACK: $ $\alpha\beta$B yz$: INPUT
- Poiché B è il non terminale più a destra in $\alpha\beta$Byz, la parte finale della handle di $\alpha\beta$Byz non può mai occorrere dentro lo stack

Viable Prefixes

- Il parser può fare lo shift di tutti i caratteri di y ed arrivare in:
- STACK: $\$\alpha\beta By$ z$: INPUT
- A questo punto può ridurre con la produzione $A \rightarrow \beta By$ ed arrivare in:
- STACK: $\$\alpha A$ z$:INPUT

Viable Prefixes

- Nel caso 2. nella configurazione:
- STACK: $\$\alpha\gamma$ xyz$: INPUT
- La handle γ è già sulla testa dello stack
- Dopo la riduzione con $B \rightarrow \gamma$ il parser può fare lo shift di tutti i simboli di xy per far apparire la nuova handle in testa allo stack:
- STACK: $\$\alpha Bxy$ z$: INPUT
- Alla fine riduce con $A \rightarrow y$

Viable Prefixes

- In entrambi i casi il parser deve fare lo shift di zero o più terminali dell'input per fare apparire sulla testa dello stack la prossima handle
- *Il parser non deve mai guardare "dentro" lo stack per trovare la handle*
- Per questo lo stack è una struttura dati adatta per fare il parsing tramite handle pruning (potatura delle handle)

Viable Prefixes

- I prefissi delle forme sentenziali destre che possono apparire sullo stack di un parser shift-reduce sono detti **viable prefixes**
- Equivalentemente, un viable prefix è un prefisso di una forma sentenziale destra che non continua mai oltre la handle più a destra della forma sentenziale
- Quindi, è sempre possibile aggiungere simboli *terminali* ad un viable prefix per ottenere una forma sentenziale destra.
- Se l'input visto fino ad un certo punto può essere ridotto ad un viable prefix allora possiamo dire che la porzione di input vista non contiene errori (apparentemente)

Conflitti

- Ci sono grammatiche libere per le quali uno shift-reduce parser non può essere usato per fare l'analisi
- Il parser raggiungerà sempre almeno uno stato in cui non può decidere, in base al contenuto dello stack e del simbolo corrente di input, se fare uno shift oppure ridurre
- In questo caso si dice che c'è un **conflitto shift/reduce**

Conflitti

- Un altro possibile conflitto può accadere quando il parser decide che bisogna fare una riduzione, ma non può decidere, in base sempre allo stack e al simbolo di lookahead, con quale produzione ridurre
- In questo caso si dice che siamo in presenza di un conflitto **reduce/reduce**
- Le grammatiche per cui il parser si può trovare in una di queste situazioni di conflitto non appartengono alla classe LR(k) (vengono chiamate anche grammatiche non LR)

Esempio

- Consideriamo la solita grammatica “difficile”:
- *stmt* → **if** *expr* **then** *stmt*

  | **if** *expr* **then** *stmt* **else** *stmt*

  | **altri-stmt**
- Supponiamo di avere un parser shift-reduce nella seguente configurazione:
- STACK: … **if** *expr* **then** *stmt*

  else … $: INPUT

Esempio

- Non possiamo dire se **if** *expr* **then** *stmt* è una handle oppure no, nemmeno guardando tutto lo stack.
- Qui c'è un conflitto shift-reduce
- In base a quello che segue l'**else** dell'input, potrebbe essere corretto ridurre **if** *expr* **then** *stmt* a *stmt*
- Ma potrebbe essere giusto anche fare lo shift dell'**else** e poi cercare uno *stmt* per chiudere il condizionale.
- Quindi la grammatica non è LR(1)

Esempio

- Nessuna grammatica ambigua può essere LR(k), per un qualsiasi k
- La grammatica dell'esempio è certamente ambigua e quindi, oltre a non essere LR(1), non esiste nessun k per il quale la grammatica è LR(k)
- Infatti, scelto un qualunque k, è possibile annidare tutti i costrutti condizionali che mi servono per riproporre il conflitto visto.

Esempio

- Comunque, in qualche caso, e questo è uno di quelli, è possibile fare il parsing della grammatica risolvendo il conflitto a favore di una delle alternative possibili.
- Se in questo esempio risolviamo il conflitto in favore dello shift abbiamo che il parser si comporta “bene” nel senso che segue la convenzione che ogni **else** viene considerato l’alternativa al **then** “pendente” (cioè senza un relativo else) che immediatamente lo precede.

LR parsing

- Introduciamo un metodo di parsing bottom-up efficiente che può essere usato per un'ampia classe di grammatiche libere
- LR(k) parsing:
  - L indica che l'input viene letto da sinistra a destra (**L**eft to right)
  - R indica che viene ricostruita una derivazione **R**ightmost rovesciata
  - k indica che vengono usati k simboli di lookahead (se k=1 spesso "(1)" viene omesso e quindi si parla semplicemente di LR parsing)

Vantaggi

- Può essere costruito un parser LR per tutti i costrutti dei linguaggi di programmazione per i quali può essere scritta una grammatica libera
- LR è il metodo di parsing shift-reduce senza backtrackin più generale che si conosca e, nonostante ciò, può essere implementato in maniera efficiente tanto quanto altri metodi di parsing shift-reduce meno generali
- La classe di grammatiche che possono essere analizzate LR propriamente maggiore di quelle che che possono essere analizzate LL con un parser predittivo

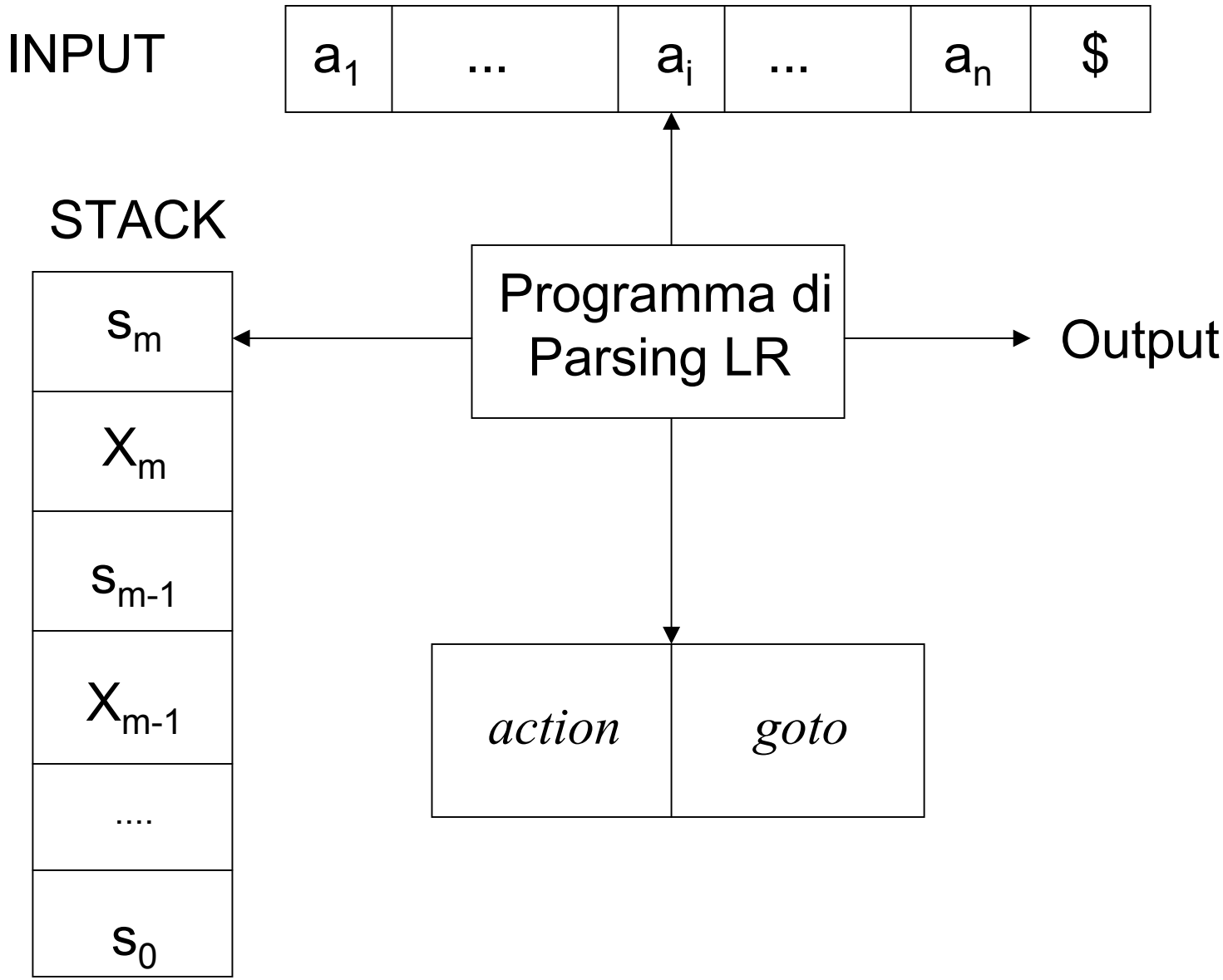
Vantaggi e svantaggi

- Tutti i parser LR rilevano un errore di sintassi il prima possibile rispetto ad una scansione dell'input da sinistra a destra
- Svantaggio: la costruzione di un parser LR per un linguaggio di programmazione tipico è troppo complicata per essere fatta a mano
- C'è bisogno di un tool apposito, un generatore di parser LR, che applichi gli algoritmi che vedremo e definisca la tabella del parser. Esempi di generatori di questo tipo sono Yacc o Bison
- Questi tool sono molto utili anche perché danno informazione diagnostica se c'è qualche problema (ad esempio, in caso di grammatica ambigua, il tool restituisce abbastanza informazione per determinare dove si crea l'ambiguità)

Programma

1. Introduciamo l'algoritmo generale eseguito da un parser LR
2. Introduciamo il metodo più semplice per la costruzione della tabella di un parser LR: simple LR (abbreviato in SLR)
3. Studiamo il metodo LR canonico che è il metodo più potente, ma anche il più costoso, per costruire la tabella
4. Semplifichiamo un po' e definiamo il metodo lookahead LR (LALR) che si trova ad un livello intermedio fra gli altri due

Il parser LR


INPUT
a_1
...
a_i
...
a_n
$
STACK
s_m
X_m
s_{m-1}
X_{m-1}
....
s_0
Programma di Parsing LR
Output
action
goto

Il parser LR

- Il programma è sempre lo stesso, cambia la tabella *action* + *goto*
- Ogni s_i nello stack è uno stato che serve a "riassumere" i simboli di stack sottostanti
- In una reale implementazione è sufficiente lo stato, non c'è bisogno di mettere anche i simboli della grammatica
- Ad ogni passo il parser decide se fare shift o reduce in base allo stato che si trova in cima allo stack e al simbolo di lookahead corrente

La tabella *action*

- Il parser consulta $action[s_m, a_i]$ che può contenere:
  - shift s, dove s è uno stato
  - reduce, con una produzione $A \rightarrow \beta$
  - accetta
  - errore

Configurazioni

- In ogni istante il parser è in una configurazione:

  $(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m,\quad a_i\ a_{i+1}\ \ldots\ a_n\ \$)$
- È la stessa configurazione tipica di uno shift-reduce parser, ma in più ci sono gli stati
- Corrisponde alla forma sentenziale destra $X_1\ X_2\ \ldots\ X_m\ a_i\ a_{i+1}\ \ldots\ a_n$

Azioni e goto

- Se *action*[s_m,a_i]= shift s allora il parser mette il simbolo a_i sullo stack
- Per arrivare ad una configurazione corretta il parser deve inserire lo stato s sopra al simbolo appena impilato
- Tale stato s è stato determinato da *goto*[s_m,a_i]=s
- La nuova configurazione è:

  (s_0 X_1 s_1 ... X_m s_m a_i s, a_{i+1} ... a_n\$)

Azioni e goto

- Se *action*[s_m, a_i] = reduce con $A \rightarrow \beta$ allora la nuova configurazione diventa:

(s_0 X_1 s_1 ... X_{m-r} s_{m-r} A s, a_i a_{i+1} ... a_n \$)

dove:

  - r è la lunghezza di β (si eliminano dallo stack 2r simboli fino ad avere in testa s_{m-r})
  - Si ha che X_{m-r+1} X_{m-r+2} ... X_m = β
  - s = *goto*[s_{m-r}, A]

Azioni e goto

- Se *action*[s_m, a_i] = accetta allora il parser annuncia che l'analisi è terminata con successo
- Se *action*[s_m, a_i] = errore allora c'è un errore di sintassi e il parser chiama una procedura di gestione/recupero dell'errore

Algoritmo

- **Input**: Una stringa di terminali (token) w ed una tabella di parsing LR (*action* + *goto*) per la grammatica considerata
- **Output**: Se w è in L(G) allora restituisce la traccia di una derivazione rightmost rovesciata di w da G, altrimenti dà una indicazione di errore

Algoritmo

ip punta al primo simbolo di w$;
while true **do begin**
    s := top(stack); a:= simbolo puntato da ip;
    if action[s,a] = shift s' **then begin**
        push(a); push(s'); ip avanza di un simbolo
    end
    else if action[s,a] = reduce $A \rightarrow \beta$ **then begin**
        pop 2*| β| simboli dallo stack; s':= top(stack);
        push(A); push(goto[s',A]);
        segnala in output la produzione $A \rightarrow \beta$
    end
    else if action[s,a] = accetta **then return**
    else errore()
end

Esempio

- Consideriamo la seguente grammatica:

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow \mathbf{id}$

- La numerazione è importante per i riferimenti della tabella

Una tabella LR per la grammatica

	action						*goto*		
Stato	**id**	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

La tabella

- Nella tabella
  - s*i* sta per “shift *i*”, cioè fai lo shift del simbolo corrente di input e poi metti sullo stack lo stato *i*
  - r*j* sta per reduce con la produzione numero *j*
  - acc significa accetta
  - vuoto significa errore
- Il valore *goto*[s,*a*] per un simbolo terminale *a* è il valore *i* di s*i*, se indicato, altrimenti è errore

Parsing della stringa: **id * id + id**

Step	STACK	INPUT	AZIONE
1	0	**id** * **id** + **id** $	Shift (s5)
2	0 **id** 5	* **id** + **id** $	Reduce con F→ **id** (r6)
3	0 F 3	* **id** + **id** $	Reduce con T → F (r4)
4	0 T 2	* **id** + **id** $	Shift (s7)
5	0 T 2 * 7	**id** + **id** $	Shift (s5)
6	0 T 2 * 7 **id** 5	+ **id** $	Reduce con F → **id** (r6)
7	0 T 2 * 7 F 10	+ **id** $	Reduce con T → T * F (r3)
8	0 T 2	+ **id** $	Reduce con E → T (r2)
9	0 E 1	+ **id** $	Shift (s6)
10	0 E 1 + 6	**id** $	Shift (s5)
11	0 E 1 + 6 **id** 5	$	Reduce con F→ **id** (r6)
12	0 E 1 + 6 F 3	$	Reduce con T → F (r4)
13	0 E 1 + 6 T 9	$	Reduce con E → E + T (r1)
14	0 E 1	$	Accetta

Costruzione delle tabelle di parsing LR

- Come costruiamo la tabella?
- Una grammatica per la quale possiamo costruirla si dice che è una grammatica LR
- Esistono grammatiche libere dal contesto che non sono grammatiche LR
- Possono essere evitate per i costrutti tipici dei linguaggi di programmazione

Intuizione

- Una grammatica è LR quando un parser shift-reduce è in grado di riconoscere le handle quando appaiono in cima allo stack
- Un parser LR non deve guardare tutto lo stack: lo stato che in ogni momento si trova in testa ad esso contiene tutta l'informazione di cui il parser ha bisogno

Intuizione

- Fatto importante: **se** è possibile riconoscere una handle guardando solo i simboli della grammaticha che sono sullo stack **allora** esite un automa finito che, leggendo i simboli dello stack dal fondo alla testa, determina se una certa handle è presente in testa
- La funzione *goto* di un parser LR è essenzialmente questo automa finito

Intuizione

- L’automa non ha bisogno di leggere tutto lo stack ad ogni mossa
- Lo stato in testa allo stack è esattamente lo stato in cui l’automa si troverebbe se lo si facesse partire dallo stato iniziale a riconoscere la stringa composta dai simboli dello stack dal fondo alla testa

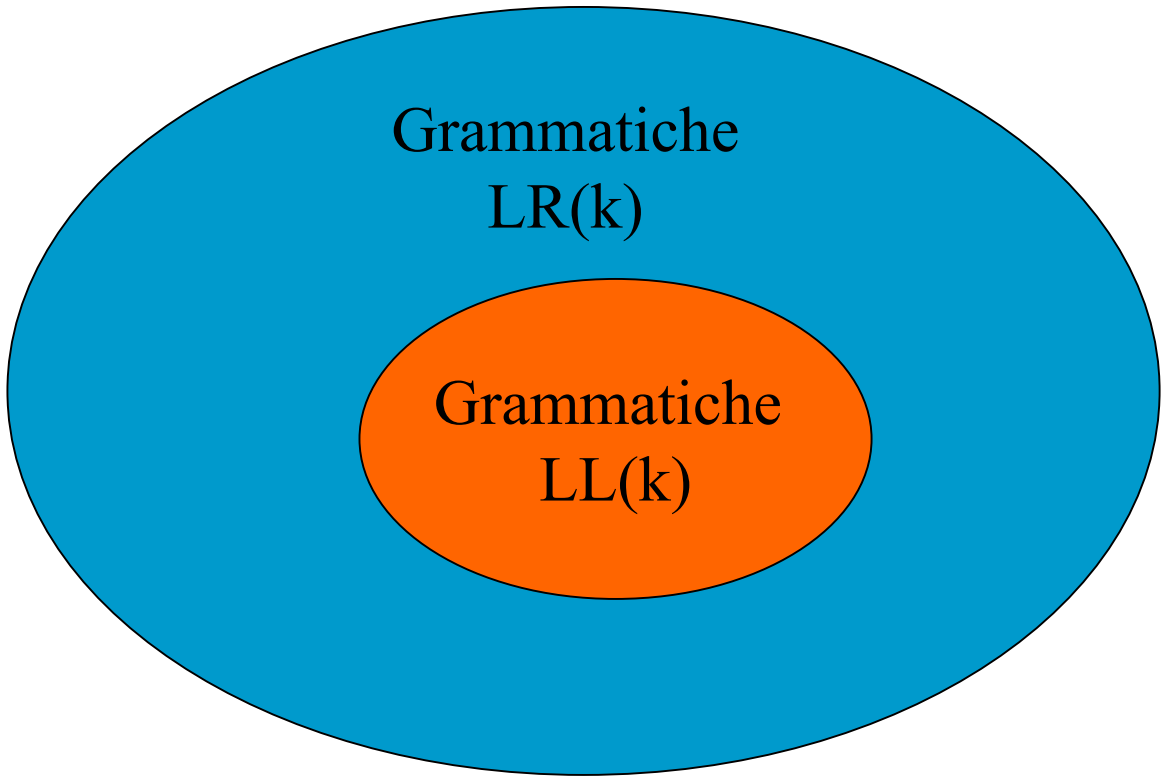
I simboli di lookahead

- Oltre allo stato in testa allo stack un parser LR prende le sue decisioni anche in base, in generale, a k simboli di lookhead
- Noi considereremo solo i casi k=0 e k=1 (che sono quelli di interesse pratico)

Grammatiche LL vs LR

- La condizione richiesta ad una grammatica per essere LR(k) è meno forte di quella richiesta alla stessa per essere LL(k)
- LR(k): dobbiamo essere in grado di riconoscere l'occorrenza della parte destra di una produzione avendo visto tutto quello che è stato derivato dalla stessa parte destra e avendo k simboli di lookahead
- LL(k): dobbiamo essere in grado di decidere quale produzione applicare ad un simbolo non terminale potendo vedere solo k simboli di quello che la parte destra della produzione in questione deriva.

Insiemi



Insiemi (k=1)