Università degli Studi di Ferrara

Riconoscitori LR(0)

Corso di Laurea Magistrale in
Ingegneria Informatica e dell'automazione

Anno accademico 2017/2018

Prof. MARCO GAVANELLI
Dipartimento di Ingegneria

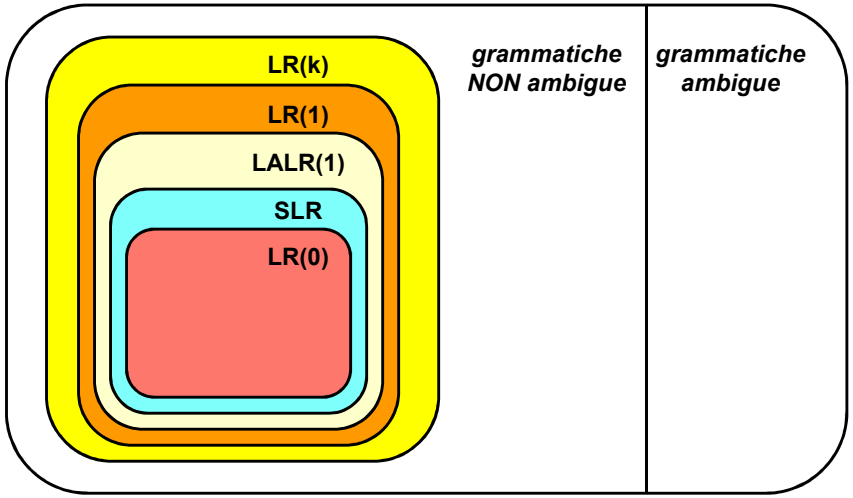
PARSING LR vs PARSING LL

- **La "debolezza" dell'analisi LL è nella sua semplicità: deve *predire* quale produzione usare avendo visto solo i primi *k* simboli della parte destra di tale produzione, *perché costruisce l'albero top-down***
- **Viceversa, l'analisi LR costruisce l'albero bottom-up: ciò le permette di *posporre le scelte fino al momento in cui ha visto abbastanza simboli da coprire tutta la parte destra della produzione + k simboli oltre***
  - **per questo, *ogni grammatica LL(k) è anche LR(k)***
- **L'analisi LR è *meno naturale* dell'analisi LL, ma è superiore dal punto di vista teorico: essa cattura infatti l'essenza del parsing deterministico left-to-right.**

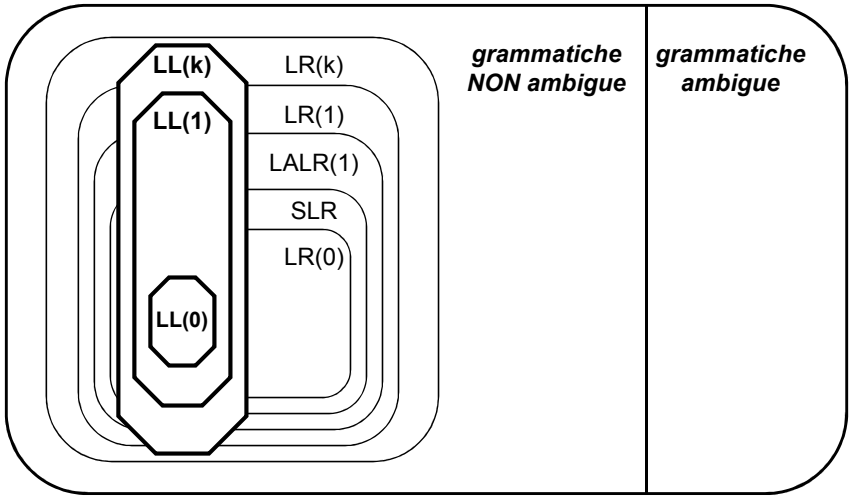
PARSING LR: FAMIGLIA

- **Sfortunatamente, l'analisi LR(1) è *talmente complessa* da risultare *spesso ingestibile* per le grammatiche dei tipici linguaggi di programmazione.**
- **Per questo motivo si definiscono *versioni semplificate* che *approssimano* la tecnica di parsing LR(1) rendendola gestibile:**
  - ***SLR*** ***(Simple LR)***
  - ***LALR(1)*** ***(Look-Ahead LR)***
- **In ogni caso, le tecniche LR sono *sempre gestite tramite strumenti automatici* per la produzione di parser, perché la costruzione manuale è «quasi impossibile».**

PARSING LR: FAMIGLIA



PARSING LR: FAMIGLIA

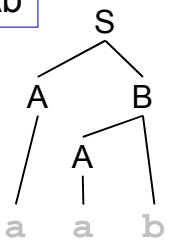


Tecniche LR

S -> AB
A -> a
B -> Ab

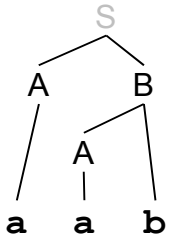
- L'analisi LL lavora top-down:
  - parte dallo scopo
  - applica le regole della grammatica come produzioni, cioè in avanti
  - si cerca di coprire la frase da riconoscere

$S \longrightarrow AB \longrightarrow aB \longrightarrow aAb \longrightarrow aab$



- L'analisi LR lavora bottom-up
  - Si parte dalla frase
  - si usano le regole della grammatica all'indietro, applicando delle riduzioni
  - si cerca di arrivare allo scopo

$aab \longrightarrow Aab \longrightarrow AAb \longrightarrow AB \longrightarrow S$

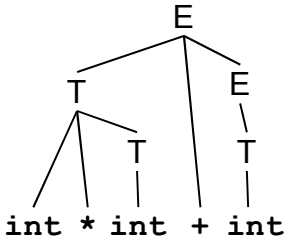


Efficienza?

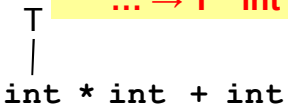
- Chiaramente, data la stringa di input si possono fare moltissime scelte diverse su quali regole utilizzare per le riduzioni
  - bisognerebbe usare un algoritmo di backtracking, ma questo avrebbe un costo proibitivo

  $E \rightarrow T + E \mid T$

  $T \rightarrow \texttt{int} * T \mid \texttt{int} \mid (E)$
- Consideriamo la stringa: `int * int + int`



Bloccato!
Non c'è una regola
… → T * int



Semplificazioni (1)

- Si parte quindi da una semplificazione: la stringa di input viene letta/esaminata da sinistra a destra
  - l'operazione di lettura di un carattere dall'input viene chiamata operazione di shift
  - Spesso si indica il punto in cui si è arrivati a leggere con |
  - Inizialmente devo ancora esaminare tutta la stringa

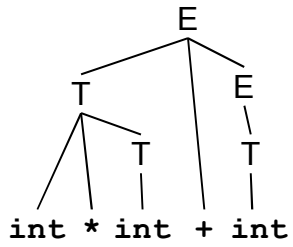
$$|x_1 x_2 \ldots x_n \ .$$

- Il parser può fare due tipi di azioni: shift e reduce.

Esempio

$E \rightarrow T + E \mid T$
$T \rightarrow \text{int} * T \mid \text{int} \mid (E)$

	|int * int + int
shift	int | * int + int
shift	int * | int + int
shift	int * int | + int
reduce T → int	int * T | + int
reduce T → int * T	T | + int
shift	T + | int
shift	T + int |
reduce T → int	T + T |
reduce E → T	T + E |
reduce E → T + E	E |



Note:
1. A destra del simbolo | c'è la parte ancora da analizzare → solo simboli terminali
2. A sinistra quando si fa una reduce si sostituisce una forma di frase subito prima di | con un nonterminale

Semplificazioni (2)

- Consideriamo solo operazioni di reduce che sostituiscono una forma di frase *subito prima* del simbolo **|** con un nonterminale
- se ho una regola $A \rightarrow \alpha$, allora posso sostituire

  $\beta\gamma\delta\alpha|abcd$

  con

  $\beta\gamma\delta A|abcd$
- ma non posso sostituire $\beta\alpha\gamma\delta|abcd$ con $\beta A\gamma\delta|abcd$
  - (avrei dovuto farlo prima, quando il cursore | era subito dopo α)
- In altre parole, la parte prima del | viene gestita come uno stack!
  - Questo ha senso: visto che stiamo parlando di grammatiche context-free, si possono riconoscere con un PDA

Interpretazione

Ma allora, visto che

- sostituisco sempre una forma di frase *'appoggiata al simbolo |'* con un nonterminale
- a destra del simbolo | ho solo terminali

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
E |

significa che se leggo i passi al contrario

- sostituisco un nonterminale *'appoggiato al simbolo |'*
- a destra di | ho solo terminali

Cioè ho una *derivazione canonica destra*

Scegliere la mossa giusta

- Resta da spiegare come fare a scegliere 'la mossa giusta'
- Data una generica situazione

  `aBcD|efg`
- come faccio a scegliere se fare
  - shift **`e`**
  - ridurre **`D`**, oppure **`cD`**, oppure **`BcD`**, oppure **`aBcD`**?

PARSER LR: ARCHITETTURA (1)

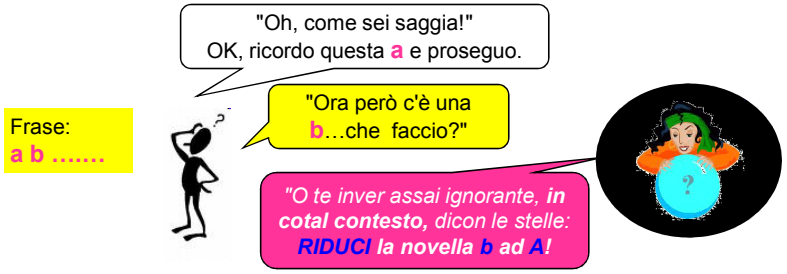
Per questo, il parser LR richiede concettualmente la presenza al suo interno di un *ORACOLO*

- **un componente che, in base al *contesto corrente*, gli dica in ogni istante se:**
  - ***proseguire la lettura dall'input*, senza fare altro → SHIFT**
  - **o *costruire un pezzo di albero* senza leggere input → REDUCE**

PARSER LR: ARCHITETTURA (2)

Per questo, il parser LR richiede concettualmente la presenza al suo interno di un *ORACOLO*

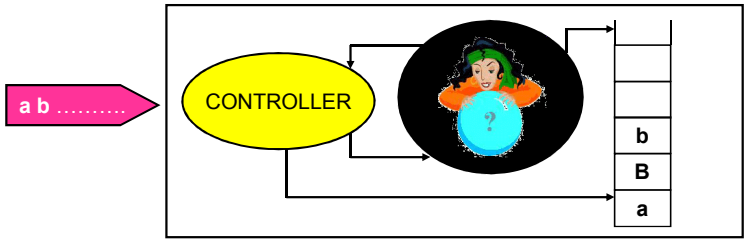
- **un componente che, in base al *contesto corrente*, gli dica in ogni istante se:**
  - ***proseguire la lettura dall'input*, senza fare altro → SHIFT**
  - **o *costruire un pezzo di albero* senza leggere input → REDUCE**



PARSER LR: ARCHITETTURA (3)

Dunque, un parser LR è costituito da:

- **un oracolo, che gli dice se fare SHIFT o REDUCE**
- **uno stack in cui conservare l'input** (e l'albero…)
- **un controller "orchestratore" che governa il tutto.**



La sequenza di riduzioni non è casuale:

- **non è altro che una *derivazione canonica destra* della frase data, ma *usata a rovescio – dalla frase allo scopo!***

L'oracolo

- Di quali informazioni dispone l'oracolo?
- Sicuramente può leggere lo stack

  aBcD|efg

- Al solito, potrà sbirciare qualche carattere 'in avanti', a seconda se è un oracolo LR(1), LR(2), … LR(k)
- Partiamo per ora da oracoli LR(0), che saranno in grado di riconoscere grammatiche LR(0)

Scegliere la mossa giusta

- In questo esempio:
  E → T + E | T
  T → int * T | int | (E)

Bloccato!
Non c'è una regola
… → T * int

```
T
|
int | *int  + int
```

- siamo rimasti bloccati
  perché a questo punto è impossibile arrivare allo scopo E della grammatica
- Intuizione: accettiamo una riduzione A → α nella situazione βα|abc solo se esiste un modo per ridurre fino al simbolo iniziale

Handles

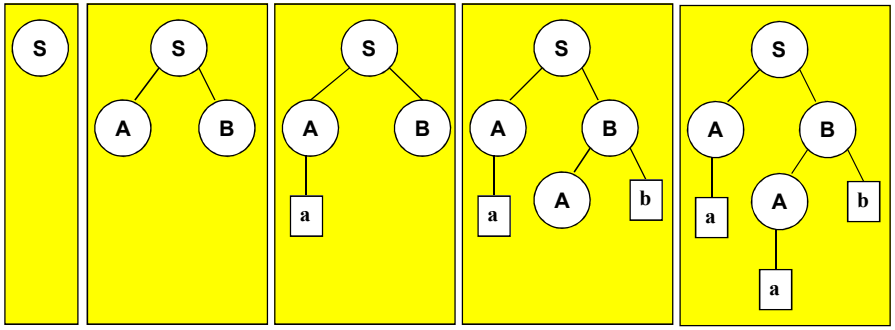
- Consideriamo una derivazione canonica destra

$$S \rightarrow^* \alpha X w \rightarrow \alpha\beta w$$

- Se esiste una tale derivazione, significa che in questo contesto ha senso ridurre β a X
- Si dice che X → β nella posizione che segue α è una handle (maniglia) di αβw

ESEMPIO: ANALISI TOP-DOWN

In un'analisi TOP-DOWN, la frase aab potrebbe essere così analizzata secondo una certa grammatica G_L :



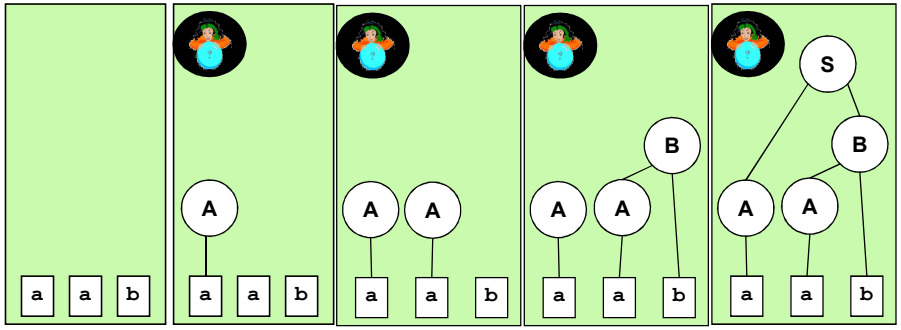
L'albero viene costruito top-down, partendo dalla radice (lo scopo S).

Ciò corrisponde ad applicare una derivazione canonica sinistra:

S → AB → aB → aAb → aab

ESEMPIO: ANALISI BOTTOM-UP

In un'analisi BOTTOM UP, la frase aab sarebbe invece riconosciuta "a rovescio", così :



Derivazione canonica destra <u>a rovescio</u>:

S ← AB ← AAb ← Aab ← aab

L'albero viene costruito bottom-up, partendo dalle foglie e risalendo.

L'INFORMAZIONE DI CONTESTO

Ma.. *COME OPERA L'ORACOLO?*

- **come fa a dire con cognizione di causa se in quel certo momento sia giusto leggere *nuovo input* (SHIFT) o effettuare invece un *passo di riduzione* (REDUCE) ?**
  - **in generale, per decidere cosa fare di una sottostringa α non basta che la grammatica contenga una produzione $A \to \alpha$: occorrono *informazioni di contesto***

- **L'oracolo è un *RICONOSCITORE DI CONTESTI***
  - **per ogni regola della grammatica, si calcolano le *informazioni di contesto***
  - **l'oracolo riconosce in che contesto ci si trova, e in base a ciò risponde**
  - **quando riconosce un contesto di riduzione, ordina la "mossa di riduzione" giusta.**

INFORMAZIONE DI CONTESTO: ESEMPIO

Sia G la grammatica seguente:

$S \to a\,A\,b \mid a\,a\,B\,b\,a$

$A \to A\,a \mid b$

$B \to \varepsilon$

G è ricorsiva a sinistra (sul simbolo A) ma ciò non è un problema per l'analisi LR.

- **Poiché Il simbolo b compare nella parte destra di *varie produzioni*, sapere che b compare nella stringa data in generale non basta per decidere quale riduzione applicare fra le tre:**

  $A \to b$ $\quad$ $S \to a\,A\,b$ $\quad$ $S \to a\,a\,B\,b\,a$

- **Ad esempio, se la stringa di input inizia con a b b …**
- **... l'azione giusta per la prima occorrenza di b è la riduzione $A \to b$,** in modo da ottenere la forma di frase aAb da cui si può proseguire, **ma per le occorrenze successive l'azione giusta sarebbe diversa!**
- **Si dice allora che la riduzione $A \to b$ è adatta alla stringa di input *nel contesto ab*, ossia se la b da ridurre è preceduta dal prefisso a.**

ANALISI LR(0)

L'idea di fondo dell'analisi LR(0) consiste nel:

- **"calcolare" *il contesto LR(0)* di ciascuna produzione**
  (ognuno può comprendere *infinite stringhe di terminali e non-terminali*)
- **verificare se ci sono *collisioni* fra i contesti di produzioni diverse,** ossia se una stringa appartenente a un contesto è un "*prefisso proprio" di una stringa di un altro (ossia, la stringa dell'altro contesto ha la prima come prefisso, a cui segue un terminale)*
- **se non ci sono collisioni, si può usare il contesto per guidare l'analisi senza dover guardare avanti**
  → la grammatica è LR(0)

Ad esempio, la grammatica precedente risulta essere LR(0):

$S \to a\,A\,b$	*contesto LR(0):*	**{ aAb }**
$S \to a\,a\,B\,b\,a$	*contesto LR(0):*	**{ aaBba }**
$A \to A\,a$	*contesto LR(0):*	**{ aAa }**
$A \to b$	*contesto LR(0):*	**{ ab }**
$B \to \varepsilon$	*contesto LR(0):*	**{ aa }**

Vedremo dopo come si ottengono

ANALISI LR(0)

L'idea di fondo dell'analisi LR(0) consiste nel:

- **"calcolare" *il contesto LR(0)* di ciascuna produzione**
  (ognuno può comprendere *infinite stringhe di terminali e non-terminali*)
- **verificare se ci sono *collisioni* fra i contesti di produzioni diverse,** ossia se una stringa appartenente a un contesto è un "*prefisso proprio" di una stringa di un altro (ossia, la stringa dell'altro contesto ha la prima come prefisso, a cui segue un terminale)*
- **se non ci sono collisioni, si può usare il contesto per guidare l'analisi senza dover guardare avanti**
  → la grammatica è LR(0)

Ad esempio, la grammatica precedente risulta essere LR(0):

$S \to a\,A\,b$	*contesto LR(0):*	**{ aAb }**
$S \to a\,a\,B\,b\,a$	*contesto LR(0):*	**{ aaBba }**
$A \to A\,a$	*contesto LR(0):*	**{ aAa }**
$A \to b$	*contesto LR(0):*	**{ ab }**
$B \to \varepsilon$	*contesto LR(0):*	**{ aa }**

NON c'è collisione **perché dopo aa c'è B (un non-terminale)**

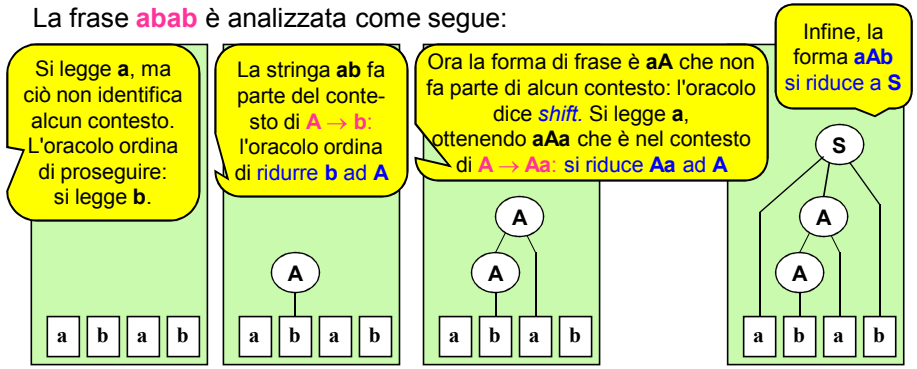
ANALISI LR(0): ESEMPIO 1

Riconsiderando la grammatica precedente:

S → a A b	*contesto LR(0):*	{ aAb }
S → a a B b a	*contesto LR(0):*	{ aaBba }
A → A a	*contesto LR(0):*	{ aAa }
A → b	*contesto LR(0):*	{ ab }
B → ε	*contesto LR(0):*	{ aa }

Vedremo dopo come si ottengono

La frase **abab** è analizzata come segue:



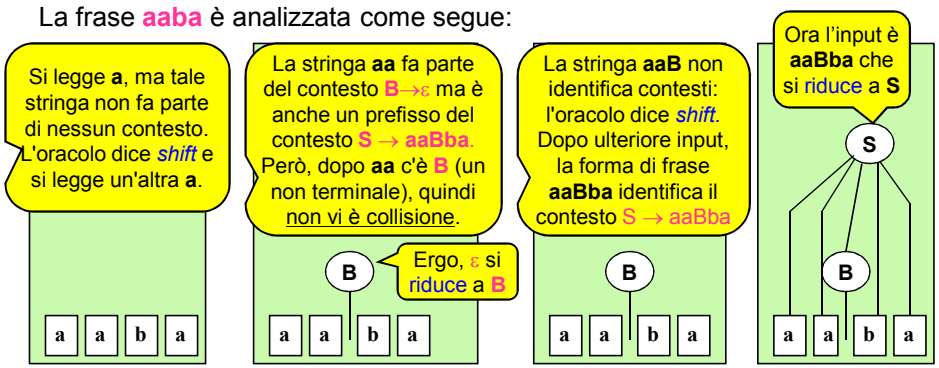
ANALISI LR(0): ESEMPIO 2

Riconsiderando la grammatica precedente:

S → a A b	*contesto LR(0):*	{ aAb }
S → a a B b a	*contesto LR(0):*	{ aaBba }
A → A a	*contesto LR(0):*	{ aAa }
A → b	*contesto LR(0):*	{ ab }
B → ε	*contesto LR(0):*	{ aa }

Vedremo dopo come si ottengono

La frase **aaba** è analizzata come segue:



ANALISI LR(0): UN CASO CRITICO

Si consideri per un istante la seguente grammatica:

S → S a	contesto LR(0):	{ Sa }
S → a	contesto LR(0):	{ a }

Apparentemente, i contesti sono diversi.

- **Tuttavia, la grammatica *non può dirsi LR(0)* perché, dopo aver ridotto la prima a a S, *non si capisce se si è realmente finito, in quanto S "riusa" se stesso*.**
  - **infatti, se non ci sono altre a, ci si è già ridotti allo scopo S**
  - **se invece la stringa continua con ulteriori a, occorre applicare ulteriori riduzione da S a ad S**
- ***Occorre "guardare avanti" di un simbolo* per capire se la stringa è finita, e dunque la grammatica non è LR(0).**
- **Per esplicitare il problema conviene *introdurre una nuova produzione Z → S dove Z è il nuovo scopo*.**

ANALISI LR(0): UN CASO CRITICO (segue)

La grammatica risulta perciò così riformulata (Z è il nuovo scopo della grammatica):

Z → S	contesto LR(0):	{ S }
S → S a	contesto LR(0):	{ Sa }
S → a	contesto LR(0):	{ a }

Ora si vede chiaramente che i primi due contesti *vanno "in collisione"*, perché S è un prefisso di Sa.

Aggiungere la nuova produzione Z → S è necessario ogni volta che lo scopo S compare anche *nella parte destra* di almeno una produzione

Ma come si calcolano i contesti LR(0)?

ANALISI LR(0): CALCOLO DEI CONTESTI

Il calcolo dei contesti LR(0) si basa sul fatto che essi sono *DEFINITI* da un'opportuna *GRAMMATICA.*

> **Proprietà essenziale:**
>
> **la grammatica che *definisce* i contesti LR(0) è sempre REGOLARE (a sinistra).**

Conseguenze

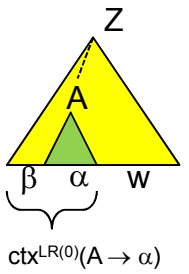
- **il riconoscimento del *contesto corrente* può essere svolto da un *automa a stati finiti***
- ***incredibile: il potentissimo ORACOLO non è altri che un opportuno automa a stati finiti!***

CONTESTI LR(0): DEFINIZIONE

Formalmente, il contesto LR(0) di una produzione $A \to \alpha$ è così definito:

$$ctx^{LR(0)}(A \to \alpha) = \{\beta\alpha \mid Z \overset{*}{\Rightarrow} \beta Aw \Rightarrow \beta\alpha w,\ w \in VT^* \}$$

- **A parole:**
  il contesto LR(0) della produzione $A \to \alpha$ è l'insieme di tutti i prefissi ($\beta\alpha$) di una forma di frase che usi la produzione $A \to \alpha$ *all'ultimo passo* ($\beta Aw \Rightarrow \beta\alpha w$) di una derivazione canonica destra.
- **Conseguenza:**
  tutte le stringhe del contesto LR(0) della produzione $A \to \alpha$ hanno la forma $\beta\alpha$ e *differiscono solo per il prefisso* β
- **ESISTE UN ALGORITMO che sfrutta questa definizione per calcolare effettivamente i contesti LR(0):** lo vedremo all'opera tra breve.
- Tuttavia, come vedremo, **è possibile - e assai più pratico - ottenere direttamente l'automa ausiliario** tramite un ***procedimento operativo.***



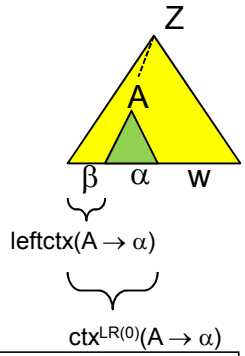
CONTESTI LR(0): CALCOLO

Poiché tutte le stringhe del contesto LR(0) della produzione $A \to \alpha$ hanno la forma $\beta\alpha$ e *differiscono solo per il prefisso* β,

si può esprimere il contesto LR(0) come *concatenazione fra un opportuno insieme e il suffisso* α

Tale "opportuno insieme" viene chiamato *contesto sinistro* del metasimbolo A.

Formalmente:



Contesto sinistro di un non-terminale A:

$$leftctx(A) = \{ \beta \mid Z \overset{*}{\Rightarrow} \beta Aw,\ w \in VT^* \}$$

onde il contesto LR(0) della produzione $A \to \alpha$ diviene:

$$ctx^{LR(0)}(A \to \alpha) = leftctx(A) \bullet \{\alpha\}$$

CALCOLO DEI CONTESTI SINISTRI (1)

Determinare leftctx(A) implica investigare *tutti i modi in cui può apparire il metasimbolo A in una forma di frase.*

Poiché lo scopo Z per definizione *non compare mai nella parte destra di alcuna produzione,* leftctx(Z) = { ε }

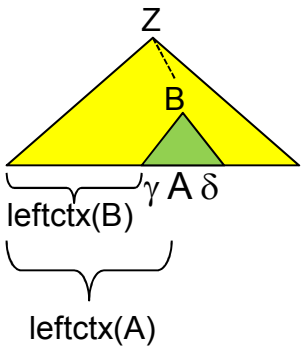
CALCOLO DEI CONTESTI SINISTRI (2)

Se c'è una produzione $B \to \gamma A \delta$, *i prefissi che possono esserci davanti ad A sono quelli che potevano esserci davanti a B seguiti dalla stringa* γ, quindi

un sottoinsieme di leftctx(A) è leftctx(B) • {γ}

ossia leftctx(A) ⊇ leftctx(B) • {γ}

L'espressione completa di leftctx(A) si ottiene perciò **unendo tutti i sottoinsiemi** ottenuti ciascuno da una diversa produ-zione in cui A compaia nella parte destra.



CONTESTI SINISTRI: ESEMPIO

Vogliamo calcolare i contesti sinistri della grammatica:

$Z \to S$

$S \to a\ S\ A\ B \mid B\ A$

$A \to a\ A \mid B$

$B \to b$

Ricordiamo le regole:

Postulato (1) :

leftctx(Z) = { ε }

Postulato (2) :

$B \to \gamma A \delta \quad \Rightarrow \quad$ leftctx(A) ⊇ leftctx(B) • {γ}

Mettiamole in pratica.

CONTESTI SINISTRI: ESEMPIO

Postulato (2) : $B \to \gamma A \delta \quad \Rightarrow \quad$ leftctx(A) ⊇ leftctx(B) • {γ}

Esempio

Postulato (1): leftctx(Z) = { ε }

Postulato (2) per la produzione $Z \to S$

leftctx(S) ⊇ leftctx(Z) • { ε } = leftctx(Z)

Postulato (2) per la produzione $S \to a\ S\ A\ B \mid B\ A$

leftctx(S) ⊇ leftctx(S) • { a } *(S → a S δ)*

leftctx(A) ⊇ leftctx(S) • { aS } *(S → a S A δ)*

leftctx(B) ⊇ leftctx(S) • { aSA } *(S → a S A B)*

leftctx(B) ⊇ leftctx(S) • { ε } *(S → B δ)*

leftctx(A) ⊇ leftctx(S) • { B } *(S → B A)*

Postulato (2) per la produzione $A \to a\ A \mid B$

leftctx(A) ⊇ leftctx(A) • { a }

leftctx(B) ⊇ leftctx(A) • { ε }

CONTESTI SINISTRI: ESEMPIO

Riordinando e ricomponendo i pezzi:

leftctx(Z) = { ε }

leftctx(S) ⊇ leftctx(Z)
leftctx(S) ⊇ leftctx(S) • { a }

leftctx(A) ⊇ leftctx(S) • { aS }
leftctx(A) ⊇ leftctx(S) • { B }
leftctx(A) ⊇ leftctx(A) • { a }

leftctx(B) ⊇ leftctx(S) • { aSA }
leftctx(B) ⊇ leftctx(S)
leftctx(B) ⊇ leftctx(A)

leftctx(Z) = { ε }

leftctx(S) = leftctx(Z) ∪
leftctx(S) • { a }

leftctx(A) = leftctx(S) • { aS } ∪
leftctx(S) • { B } ∪
leftctx(A) • { a }

leftctx(B) = leftctx(S) • { aSA } ∪
leftctx(S) ∪
leftctx(A)

Dà luogo a una grammatica regolare!

DAI CONTESTI ALLA GRAMMATICA REGOLARE AUSILIARIA

leftctx(Z) = { ε }		⟨LctxZ⟩ → ε
leftctx(S) =	leftctx(Z) ∪ leftctx(S) • { a }	⟨LctxS⟩ → ⟨LctxZ⟩ | ⟨LctxS⟩ a
leftctx(A) =	leftctx(S) • { aS } ∪ leftctx(S) • { B } ∪ leftctx(A) • { a }	⟨LctxA⟩ → ⟨LctxS⟩ aS | ⟨LctxS⟩ B | ⟨LctxA⟩ a
leftctx(B) =	leftctx(S) • { aSA } ∪ leftctx(S) ∪ leftctx(A)	⟨LctxB⟩ → ⟨LctxS⟩ aSA | ⟨LctxS⟩ | ⟨LctxA⟩

I simboli (terminali e non) della grammatica originale sono i simboli **terminali** di questa grammatica

DALLA GRAMMATICA REGOLARE AUSILIARIA AI CONTESTI SINISTRI

Risolvendo le equazioni, si possono quindi ottenere le espressioni regolari dei vari contesti:

leftctx(Z) = { ε }

leftctx(S) = { a* }

leftctx(A) = { (a* a S + a* B) a* }

leftctx(B) = { a* a S A + a* + (a* a S + a* B) a* }

⟨LctxZ⟩ → ε

⟨LctxS⟩ → ⟨LctxZ⟩ | ⟨LctxS⟩ a

⟨LctxA⟩ → ⟨LctxS⟩ aS | ⟨LctxS⟩ B | ⟨LctxA⟩ a

⟨LctxB⟩ → ⟨LctxS⟩ aSA | ⟨LctxS⟩ | ⟨LctxA⟩

DAI CONTESTI SINISTRI AI CONTESTI LR(0)

Espressioni regolari semplificate:

leftctx(Z) = { ε }

leftctx(S) = { a* }

leftctx(A) = { a* (a S + B) a* }

leftctx(B) = { a* (a S A + ε)+ a* (a S + B) a* }

Concatenando i contesti sinistri con i rispettivi suffissi:

$ctx^{LR(0)}(Z \to S) = \{ \varepsilon \}\ S = \{ S \}$

$ctx^{LR(0)}(S \to aSAB) = \{ a^*aSAB \}$

$ctx^{LR(0)}(S \to BA) = \{ a^*BA \}$

$ctx^{LR(0)}(A \to aA) = \{ a^*\ (a\ S + B)\ a^*\ aA \}$

$ctx^{LR(0)}(A \to B) = \{ a^*\ (a\ S + B)\ a^*\ B \}$

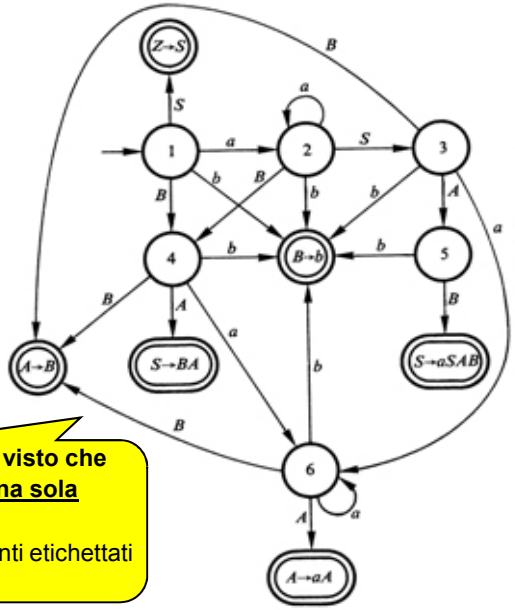
$ctx^{LR(0)}(B \to b) = \{ a^*\ (a\ S\ A + \varepsilon\ )b + a^*\ (a\ S + B)\ a^*\ b \}$

Ora si può costruire un RSF che riconosca *l'unione* di questi linguaggi: se risulta deterministico (ossia fra i contesti LR(0) non ci sono conflitti), **il gioco è fatto.**

L'AUTOMA A STATI AUSILIARIO

Questo RSF è la *macchina caratteristica* della grammatica iniziale:

- **ogni stato finale è etichettato con una produzione, quella che il parser LR deve usare in questo contesto !**
- **Si riparte ogni volta dallo stato iniziale e si percorre l'ASF per sapere in ogni momento che mossa fare.**



E' andata bene! E' deterministico, visto che
- ogni stato finale è etichettato da **una sola** produzione
- nessuno stato finale ha archi uscenti etichettati da simboli terminali.

IL PARSING LR(0) ALL'OPERA

L'analisi LR(0) di una frase si svolge appoggiandosi all'automa caratteristico e alternando due operazioni:

- **SHIFT: quando si usa un simbolo della forma di frase corrente per cambiare stato all'automa caratteristico**
- **REDUCE: quando l'automa caratteristico raggiunge uno stato finale e quindi si applica quella produzione per effettuare un passo di riduzione. In questo caso, *l'input non viene letto*.**

Dopo ogni riduzione l'automa ricomincia dallo stato 1.

Per comodità, la forma di frase corrente è mantenuta su uno stack

- **SHIFT pone in cima allo stack il nuovo simbolo terminale letto**
- **REDUCE toglie tanti elementi quanti ne corrispondono alla parte destra della riduzione da applicare, poi pone in cima il simbolo non-terminale corrispondente alla parte sinistra di tale riduzione.**

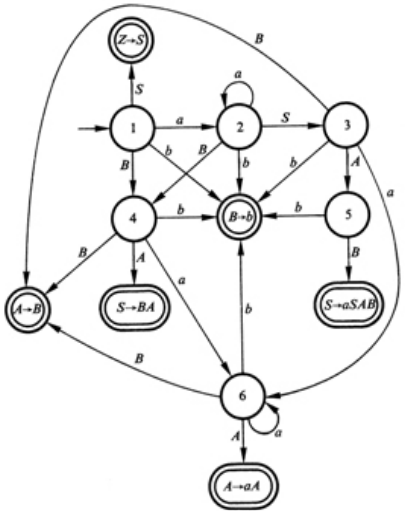
ESEMPIO DI PARSING BOTTOM-UP LR(0)

ESEMPIO: parsing bottom-up di abbabb

1/3

- simbolo: a stato: 1 stato futuro: 2
  simbolo: b stato: 2 stato finale: B → b
  riduzione: abbabb → aBbabb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: B stato: 2 stato futuro: 4
  simbolo: b stato: 4 stato finale: B → b
  riduzione: aBbabb → aBBabb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: B stato: 2 stato futuro: 4
  simbolo: B stato: 4 stato finale: A → B
  riduzione: aBBabb → aBAabb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: B stato: 2 stato futuro: 4
  simbolo: A stato: 4 stato finale: S → BA
  riduzione: aBAabb → aSabb

segue



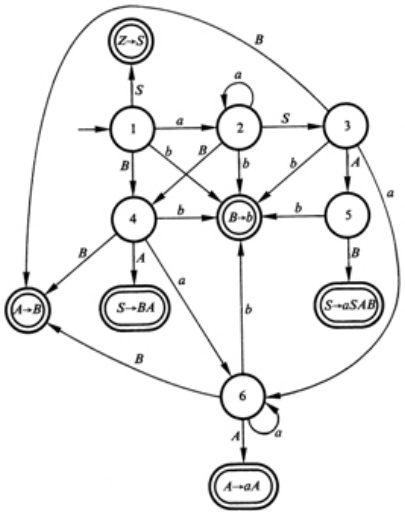
ESEMPIO DI PARSING BOTTOM-UP LR(0)

ESEMPIO (segue): parsing di aSabb

2/3

- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: a stato: 3 stato futuro: 6
  simbolo: b stato: 6 stato finale: B → b
  riduzione: aSabb → aSaBb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: a stato: 3 stato futuro: 6
  simbolo: B stato: 6 stato finale: A → B
  riduzione: aSaBb → aSaAb
- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: a stato: 3 stato futuro: 6
  simbolo: A stato: 6 stato finale: A → aA
  riduzione: aSaAb → aSAb

segue



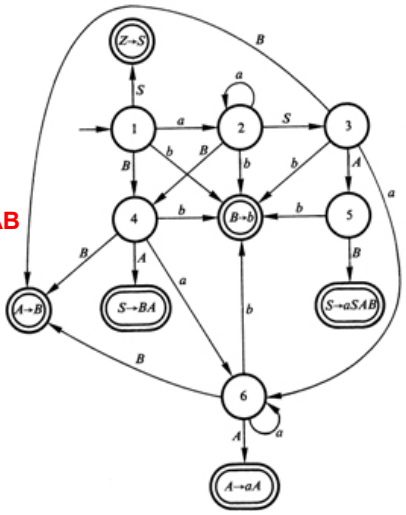
ESEMPIO DI PARSING BOTTOM-UP LR(0)

ESEMPIO (segue): parsing di aSAb

3/3

- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: A stato: 3 stato futuro: 5
  simbolo: b stato: 5 stato finale: B → b
  riduzione: aSAb → aSAB
- simbolo: a stato: 1 stato futuro: 2
  simbolo: S stato: 2 stato futuro: 3
  simbolo: A stato: 3 stato futuro: 5
  simbolo: B stato: 5 stato finale: S → aSAB
  riduzione: aSAB → S
- simbolo: S stato: 1 stato finale: Z → S
  riduzione: S → Z

frase ridotta allo scopo: OK



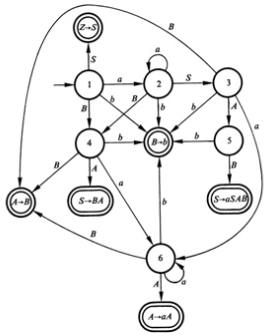
EVOLUZIONE DELLO STACK

Parsing di abbabb

EVOLUZIONE TEMPORALE DELLO STACK

- quando lo stack aumenta con un nuovo simbolo terminale → ***shift***
- quando si toglie dallo stack una sequenza (di terminali e non terminali) sostituendola con un non-terminale → ***reduce***

L'automa caratteristico:



							b	B	A		b	B		
			b	B	A		a	a	a	A	A	A		
	b	B	B	B	B	S	S	S	S	S	S	S		
a	a	a	a	a	a	a	a	a	a	a	a	a	S	Z

MIGLIORAMENTI

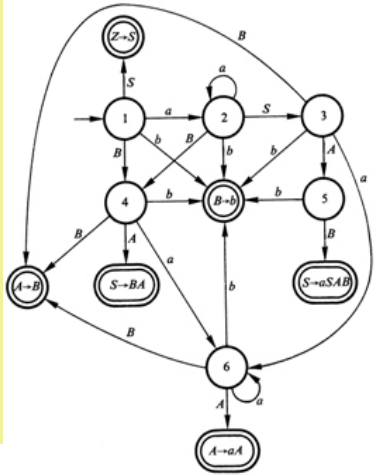
- **Nell'esempio, ogni volta si riparte dallo stato iniziale e si ripercorre l'ASF dall'inizio.**
- **In realtà, ciò non è necessario, perché *la prima parte del percorso ogni volta coincide con la precedente:* cambia solo l'ultima parte del percorso.**
- **Perciò, conviene ricordarsi il percorso e, al "giro" successivo, ripartire dall'ultimo "stato utile", non da capo**
  - **si evita così anche di dover rileggere i primi simboli della stringa di input (che potrebbero essere stati consumati)**
- **Conviene dunque avere uno stack degli stati, in cui *impilare via via i vari stati attraversati (push),* e poi *disimpilarne tanti quanti i simboli (terminali e non) coinvolti in un passo di riduzione.***

L'ESEMPIO DI PARSING OTTIMIZZATO

1/3

ESEMPIO: parsing di abbabb

- stack degli stati: [1] (stato iniziale)
  simbolo: a stato: 1 stato futuro: 2
  simbolo: b stato: 2 stato finale: B → b
  stack degli stati: [1, 2, B→b]
  riduzione: abbabb → aBbabb
  stack degli stati: [1, 2] (top: 2)
- simbolo: B stato: 2 stato futuro: 4
  simbolo: b stato: 4 stato finale: B → b
  stack degli stati: [1, 2, 4, B→b]
  riduzione: aBbabb → aBBabb
  stack degli stati: [1, 2, 4] (top: 4)
- simbolo: B stato: 4 stato finale: A → B
  stack degli stati: [1, 2, 4, A→B]
  riduzione: aBBabb → aBAabb
  stack degli stati: [1, 2, 4] (top: 4)



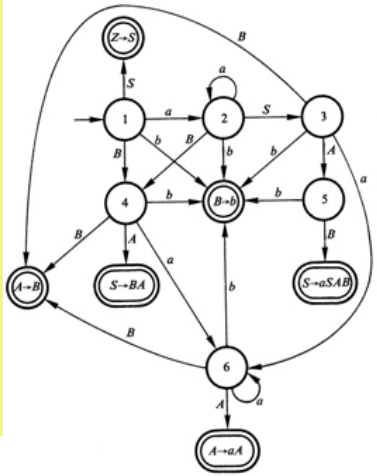
L'ESEMPIO DI PARSING OTTIMIZZATO

2/3

ESEMPIO (segue): parsing di aBAabb

- stack degli stati: [1, 2,4] (top: 4)
  simbolo: A stato: 4 stato finale: S → BA
  stack degli stati: [1, 2, 4, S→BA]
  riduzione: aBAabb → aSabb
  stack degli stati: [1, 2] (top: 2)
- simbolo: S stato: 2 stato futuro: 3
  simbolo: a stato: 3 stato futuro: 6
  simbolo: b stato: 6 stato finale: B → b
  stack degli stati: [1, 2, 3, 6, B→b]
  riduzione: aSabb → aSaBb
  stack degli stati: [1, 2, 3, 6] (top: 6)
- simbolo: B stato: 6 stato finale: A → B
  stack degli stati: [1, 2, 3, 6, A→B]
  riduzione: aSaBb → aSaAb
  stack degli stati: [1, 2, 3, 6] (top: 6)

segue

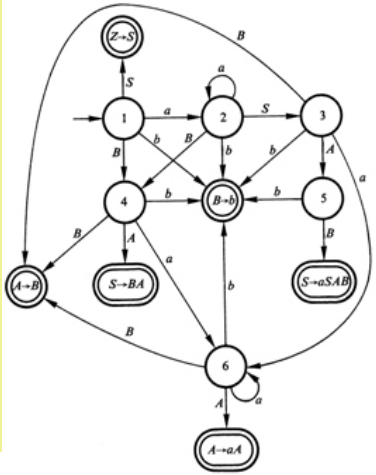


L'ESEMPIO DI PARSING OTTIMIZZATO

ESEMPIO (segue): parsing di aSaAb

- stack degli stati: [1, 2, 3, 6] (top: 6)
  simbolo: A stato: 6 stato finale: A → aA
  stack degli stati: [1, 2, 3, 6, A→aA]
  riduzione: aSaAb → aSAb
  stack degli stati: [1, 2, 3] (top: 3)
- simbolo: A stato: 3 stato futuro: 5
  simbolo: b stato: 5 stato finale: B → b
  stack degli stati: [1, 2, 3, 5, B→b]
  riduzione: aSAb → aSAB
  stack degli stati: [1, 2, 3, 5] (top: 5)
- simbolo: B stato: 5 stato finale: S → aSAB
  stack degli stati: [1, 2, 3, 5, S→aSAB]
  riduzione: aSAB → S
  stack degli stati: [1] (top: 1)
- simbolo: S stato: 1 stato finale: Z → S
  stack degli stati: [1, Z→S]
  riduzione: S → Z
  stack degli stati: [1] (situaz. iniziale: OK)

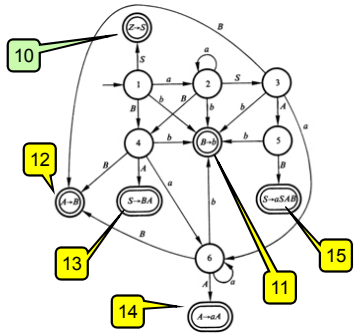
3/3



EVOLUZIONE DEGLI STACK (1/3)

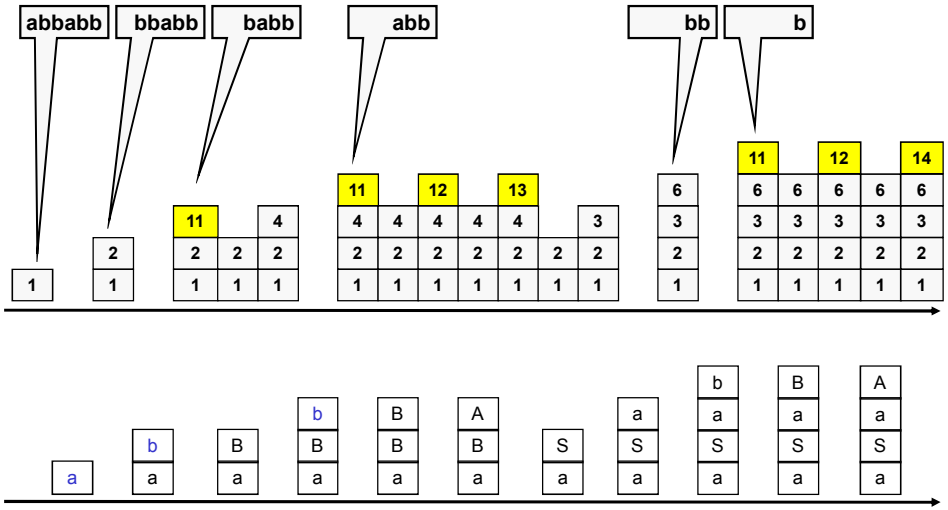
- **All'inizio,**
  - lo **stack degli stati** contiene solo lo stato iniziale, 1
- **Dopo ogni lettura dell'input,**
  - si pone sullo **stack degli stati** (push) lo stato corrisp. alla transizione
  - si pone sullo **stack di input** (push) il simbolo appena letto
- **A ogni passo di riduzione,**
  - si estraggono dallo **stack degli stati** (pop) tanti stati quanti i simboli coinvolti nella parte destra della riduzione applicata e si pone sullo stack lo stato risultante dalla riduzione.
  - contestualmente si estraggono dallo **stack di input** i simboli coinvolti nella parte destra della riduzione applicata e si pone su tale stack il metasimbolo risultante dalla riduzione.

L'automa caratteristico:

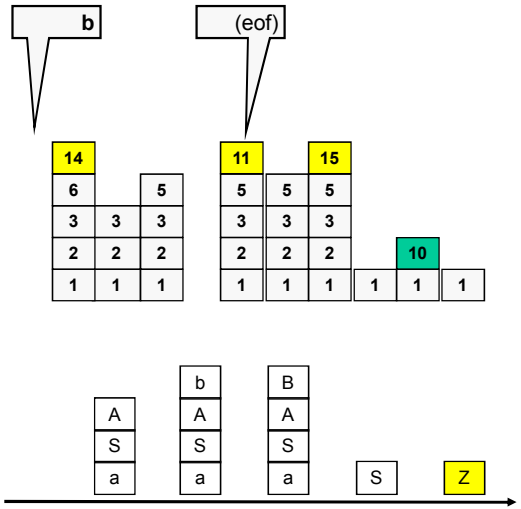


EVOLUZIONE DEGLI STACK (2/3)

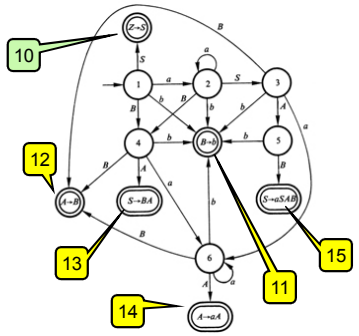
stringa ancora disponibile sullo stream di input



EVOLUZIONE DEGLI STACK (3/3)



L'automa caratteristico:



CONDIZIONE LR(0)

QUANDO È CHE UNA GRAMMATICA È ANALIZZABILE MEDIANTE ANALISI LR(0) ?

CONDIZIONE LR(0)

Condizione SUFFICIENTE perché una grammatica sia LR(0) è che, date le due produzioni $A \to \alpha$ e $B \to \omega$

- **se** $\theta \in ctx^{LR(0)}(A \to \alpha)$ $\theta \in (VT \cup VN)^*$
- **e** $\theta w \in ctx^{LR(0)}(B \to \omega)$ $w \in VT^*$

deve essere

$$w = \varepsilon, \quad A = B, \quad \alpha = \omega$$

A parole:

ogni **stato di riduzione** dell'automa ausiliario dev'essere etichettato da una produzione unica, e NON deve avere archi di uscita etichettati da simboli terminali.

CONDIZIONE LR(0): controllo

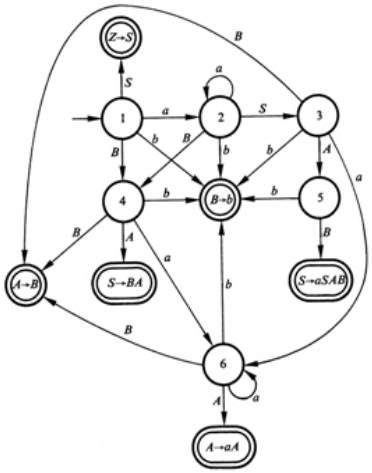
Condizione LR(0)

ogni stato di riduzione dell'automa caratteristico dev'essere etichettato da una produzione unica, e NON deve avere archi di uscita etichettati da simboli terminali.

Si può verificare che *l'automa a lato verifica la condizione sufficiente LR(0).*

Quindi la grammatica di partenza è analizzabile con analisi LR(0).

Teorema: una grammatica LR(0) non è mai ambigua.

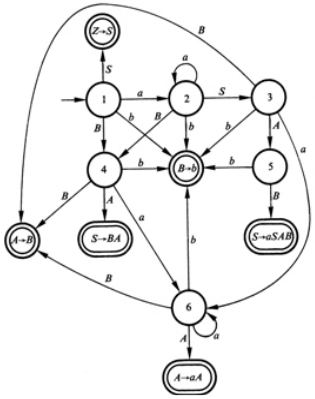


COSTRUZIONE DELL'AUTOMA CARATTERISTICO

Problema: il procedimento seguito fin qui per costruire l'automa caratteristico è stato molto lungo, complesso e non facilmente meccanizzabile.

Serve un procedimento più snello

Proprietà:
L'automa caratteristico può essere ottenuto *senza calcolare né i contesti sinistri né i contesti LR(0),* applicando in alternativa un *procedimento operativo meccanizzabile.*



PROCEDIMENTO OPERATIVO (1/10)

Illustreremo il procedimento con un esempio passo-passo.

CONVENZIONI

1) Ogni frase lecita è esplicitamente terminata dal terminatore $
Quindi, la produzione di top-level aggiunta è $Z \to S \$$

2) Data una forma di frase, il cursore . denota il *confine* tra la parte già analizzata (a sinistra) **e quella ancora da analizzare** (a destra)

- **Esempio:** $A \to .aB$ *il prossimo input è a*
- **Esempio:** $A \to a.B$ *il prossimo input è B*

dove

- per "input" si intende la forma di frase eventualmente già sottoposta a passi di riduzione, quindi costituita *sia da simboli terminali sia da non-terminali* della grammatica di partenza.
- l'automa si appoggia a uno stack di input: i simboli alla sinistra del cursore sono già stati estratti dallo stack, quelli alla destra devono ancora esserlo. Il "prossimo input" è il simbolo al top dello stack.

PROCEDIMENTO OPERATIVO (2/10)

Facciamo riferimento alla "solita" grammatica:

$Z \to S\,\$$ $\quad S \to a\,S\,A\,B \mid B\,A$

$A \to a\,A \mid B$ $\quad B \to b$

- **All'inizio, tutto l'input è ancora da leggere**
- **Inoltre, ogni frase lecita deriva dallo scopo Z**
- **Quindi, la situazione è schematizzabile scrivendo**

$$Z \to .\,S\$$$

- **Poiché a destra del cursore c'è S, le frasi possono iniziare solo con produzioni di S**
- **Ergo, per descrivere compiutamente questo stato occorre anche aggiungere tutte le produzioni di S:**

$$Z \to .\,S\$$$
$$S \to .\,aSAB \mid .\,BA$$

- **e, poiché una di queste produzioni può iniziare per B, anche**

$$B \to .\,b$$

PROCEDIMENTO OPERATIVO (3/10)

Facciamo riferimento alla "solita" grammatica:

$Z \to S\,\$$ $\quad S \to a\,S\,A\,B \mid B\,A$

$A \to a\,A \mid B$ $\quad B \to b$

Quindi, il primo stato dell'automa caratteristico, che rappresenta tutte le situazioni iniziali possibili, è descritto da:

Lo chiameremo stato 1.

$$Z \to .\,S\$$$
$$S \to .\,aSAB \mid .\,BA$$
$$B \to .\,b$$

(1)

Ora dobbiamo investigare le possibili evoluzioni, ossia **"spostare il cursore a destra"** di una posizione ***in tutti i modi possibili:***

- se il prossimo input è S, la situazione successiva sarà $Z \to S\,.\,\$$ (F)
- se il prossimo input è a, la situazione successiva sarà $S \to a\,.\,SAB$ (2)
- se il prossimo input è B, la situazione successiva sarà $S \to B\,.\,A$ (4)
- se il prossimo input è b, la situazione successiva sarà $B \to b\,.$ (R1)

PROCEDIMENTO OPERATIVO (4/10)

Facciamo riferimento alla "solita" grammatica:

$Z \to S\,\$$ $\quad S \to a\,S\,A\,B \mid B\,A$

$A \to a\,A \mid B$ $\quad B \to b$

Due di questi stati sono già finali:

- la situazione $Z \to S\,.\,\$$ rappresenta lo stato finale di accettazione, F (F)
- la situazione $B \to b\,.$ rappresenta uno stato finale di riduzione, R1: in tale stato l'automa dovrà applicare il passo di riduzione $B \to b$. (R1)

- **Nella situazione $S \to a\,.\,SAB$ (stato 2), l'input può iniziare per S, quindi vanno considerate anche tutte le produzioni relative a S; poiché fra queste ultime una può iniziare per B, va aggiunta anche la produzione $B \to b$**

$$S \to a\,.\,SAB$$
$$S \to .\,aSAB \mid .\,BA$$
$$B \to .\,b$$

(2)

PROCEDIMENTO OPERATIVO (5/10)

Facciamo riferimento alla "solita" grammatica:

$Z \to S\,\$$ $\quad S \to a\,S\,A\,B \mid B\,A$

$A \to a\,A \mid B$ $\quad B \to b$

- **Analogamente, nella situazione $S \to B\,.\,A$ (stato 4), l'input può iniziare per A, quindi vanno considerate anche tutte le produzioni relative a A; poiché fra queste ultime una può iniziare per B, va aggiunta anche la produzione $B \to b$.**

$$S \to B\,.\,A$$
$$A \to .\,aA \mid .\,B$$
$$B \to .\,b$$

(4)

Di nuovo, dobbiamo ora investigare tutte le possibili evoluzioni degli stati 2 e 4 così ottenuti.

PROCEDIMENTO OPERATIVO (6/10)

$S \to a \,.\, SAB$	(2)
$S \to \,.\, aSAB \mid \,.\, BA$	
$B \to \,.\, b$	

$S \to B \,.\, A$	(4)
$A \to \,.\, aA \mid \,.\, B$	
$B \to \,.\, b$	

Dallo stato 2:

- se il prossimo input è **S**, la situazione successiva sarà $S \to aS \,.\, AB$ (3)
- se il prossimo input è **a**, la situazione successiva sarà $S \to a \,.\, SAB$ (2)
- se il prossimo input è **B**, la situazione successiva sarà $S \to B \,.\, A$ (4)
- se il prossimo input è **b**, la situazione successiva sarà $B \to b \,.$ (R1)

Dallo stato 4:

- se il prossimo input è **A**, la situazione successiva sarà $S \to BA \,.$ (R3)
- se il prossimo input è **a**, la situazione successiva sarà $A \to a \,.\, A$ (6)
- se il prossimo input è **B**, la situazione successiva sarà $A \to B \,.$ (R2)
- se il prossimo input è **b**, la situazione successiva sarà $B \to b \,.$ (R1)

PROCEDIMENTO OPERATIVO (7/10)

Facciamo riferimento alla "solita" grammatica:

$Z \to S\ \$$ $\qquad$ $S \to a\ S\ A\ B \mid B\ A$

$A \to a\ A \mid B$ $\qquad$ $B \to b$

Analogamente analizziamo gli stati 3 e 6:

- **Nella situazione $S \to aS \,.\, AB$ (stato 3), l'input può iniziare per A, quindi vanno aggiunte le produzioni di A e la solita $B \to b$.**
- **Nella situazione $A \to a \,.\, A$ (stato 6), l'input può iniziare solo per A, quindi vanno aggiunte le produzioni di A e la solita $B \to b$.**

$S \to aS \,.\, AB$	(3)
$A \to \,.\, aA \mid \,.\, B$	
$B \to \,.\, b$	

$A \to a \,.\, A$	(6)
$A \to \,.\, aA \mid \,.\, B$	
$B \to \,.\, b$	

PROCEDIMENTO OPERATIVO (8/10)

$S \to aS \,.\, AB$	(3)
$A \to \,.\, aA \mid \,.\, B$	
$B \to \,.\, b$	

$A \to a \,.\, A$	(6)
$A \to \,.\, aA \mid \,.\, B$	
$B \to \,.\, b$	

Dallo stato 3:

- se il prossimo input è **A**, la situazione successiva sarà $S \to aSA \,.\, B$ (5)
- se il prossimo input è **a**, la situazione successiva sarà $A \to a \,.\, A$ (6)
- se il prossimo input è **B**, la situazione successiva sarà $A \to B \,.$ (R2)
- se il prossimo input è **b**, la situazione successiva sarà $B \to b \,.$ (R1)

Dallo stato 6:

- se il prossimo input è **A**, la situazione successiva sarà $A \to aA \,.$ (R4)
- se il prossimo input è **a**, la situazione successiva sarà $A \to a \,.\, A$ (6)
- se il prossimo input è **B**, la situazione successiva sarà $A \to B \,.$ (R2)
- se il prossimo input è **b**, la situazione successiva sarà $B \to b \,.$ (R1)

PROCEDIMENTO OPERATIVO (9/10)

Facciamo riferimento alla "solita" grammatica:

$Z \to S\ \$$ $\qquad$ $S \to a\ S\ A\ B \mid B\ A$

$A \to a\ A \mid B$ $\qquad$ $B \to b$

Resta da analizzare l'ultimo stato, 5:

- **Nella situazione $S \to aSA \,.\, B$, l'input può iniziare solo per B, quindi va aggiunta semplicemente la solita produzione $B \to b$.**

$S \to aSA \,.\, B$	(5)
$B \to \,.\, b$	

Dallo stato 5:

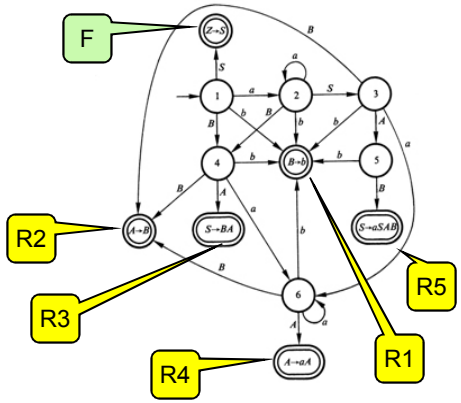
- se il prossimo input è **B**, la situazione successiva sarà $S \to aSAB \,.$ (R5)
- se il prossimo input è **b**, la situazione successiva sarà $B \to b \,.$ (R1)

Finalmente possiamo disegnare l'automa completo...

PROCEDIMENTO OPERATIVO (10/10)

… e il risultato non dovrebbe stupire:

È esattamente lo stesso di prima!



PARSER LR(0): COSTRUZIONE

Tecnicamente:

- **ogni arco corrisponde a un'azione *shift***
  detta più precisamente ***goto*** se l'arco corrisponde a un **metasimbolo**
  - la distinzione è utile perché nel caso *shift* occorre prelevare un nuovo simbolo terminale, nell'altro un metasimbolo - e a volte sono su stack separati.
- **ogni stato finale corrisponde a un'azione *reduce***
  detta più precisamente ***accept*** nel caso $Z \rightarrow S\,.\,\$$

Si costruisce una *tabella di parsing*

- **è simile alla tabella delle transizioni di un qualunque automa…**
- **… ma, oltre allo stato futuro, indica anche *l'azione da compiere***

Anziché scrivere semplicemente S6 per dire "vai nello stato 6", si scriverà quindi *un'informazione più articolata*, ad esempio qualcosa del tipo:

- ***s / 5*** per dire "fai uno **shift sulla stringa di input e vai nello stato 5**"
- ***g / 5*** per dire "**consuma il metasimbolo corrente e vai nello stato 5**"
- ***r / 4*** per dire **"riduci usando la produzione n° 4" (e termina)**
- ***a*** per dire "**accetta**" **(e termina)**

PARSER LR(0): TABELLA DI PARSING

Algoritmo di costruzione della *tabella di parsing* :

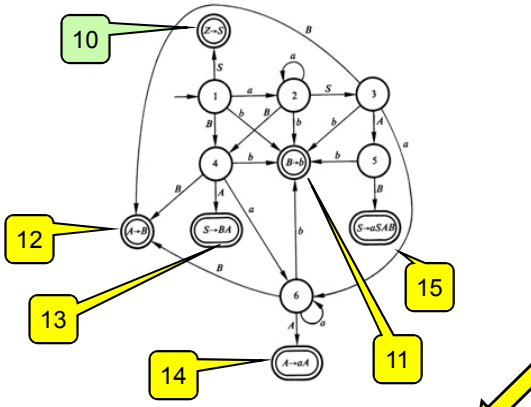
- per ogni arco **S1 → S2 con input il simbolo terminale *a*,** si inserisce in tabella l'azione ***shift to S2* alla posizione (S1, *a*)**
- per ogni arco **S1 → S2 con input il metasimbolo *X*,** si inserisce in tabella l'azione ***goto S2* alla posizione (S1, *X*)**
- per ogni stato S_i **associato alla produzione k-esima,** $A \rightarrow \alpha\,.$ si inserisce in tabella l'azione ***reduce k* in tutta la riga corrispondente allo stato S_i**
- per ogni stato S_i **contenente la produzione** $Z \rightarrow S\,.\,\$$ si inserisce in tabella l'azione ***accept* alla posizione (S_i, $)**

Ovviamente, *molte posizioni resteranno vuote,* poiché si omettono tutte le situazioni di errore. La parsing table è quindi *sparsa.*

Applichiamo l'algoritmo all'automa dell'esempio precedente.

TABELLA DI PARSING: ESEMPIO

L'automa con gli stati rinumerati:



$B \rightarrow b$ (1) $A \rightarrow B$ (2)
$S \rightarrow B\,A$ (3) $A \rightarrow a\,A$ (4)
$S \rightarrow a\,S\,A\,B$ (5)

	a	b	$	S	A	B
1	s2	s11		g10		g4
2	s2	s11		g3		g4
3	s6	s11			g5	g12
4	s6	s11			g13	g12
5		s11				g15
6	s6				g14	g12
10			a			
11	r1	r1	r1			
12	r2	r2	r2			
13	r3	r3	r3			
14	r4	r4	r4			
15	r5	r5	r5			

PARSER LR(0): CONSIDERAZIONI

Un parser LR(0) sa sempre se fare shift o reduce *guardando solo lo stato attuale*: *se opta per ridurre, non esegue shift e quindi non legge alcun simbolo dall'input.*

SOLO SE ha deciso di fare shift (cioè di leggere dall'input un altro simbolo) sfrutta tale simbolo per decidere lo stato futuro.

Per questo, in corrispondenza a uno stato di riduzione, l'azione è necessariamente la stessa in tutte le colonne dei simboli terminali: l'input non è stato letto!
Qui si constata l'essenza dell' LR(0)

	a	b	$	S	A	B
1	s2	s11		g10		g4
2	s2	s11		g3		g4
3	s6	s11			g5	g12
4	s6	s11			g13	g12
5		s11				g15
6	s6				g14	g12
10			a			
11	r1	r1	r1			
12	r2	r2	r2			
13	r3	r3	r3			
14	r4	r4	r4			
15	r5	r5	r5			

Legenda

stati di SHIFT	stati di GOTO
stati di RIDUZIONE	stato di ACCEPT

CONTROESEMPIO 1

Si consideri la seguente grammatica:
$Z \to E$ $E \to T+E \mid T$ $T \to a$

Questa grammatica presenta ricorsione destra.

Per vedere se è LR(0), si può:

- **o calcolare i contesti LR(0) e verificare se rispettano la *condizione sufficiente* già definita**
- **o applicare il procedimento e vedere se l'automa caratteristico è deterministico o, invece, presenta *conflitti.***

Come esercizio, seguiremo entrambe le vie, ponendole poi a confronto.

CONTROESEMPIO 1: calcolo contesti (1/3)

Si consideri la seguente grammatica:
$Z \to E$ $E \to T+E \mid T$ $T \to a$

$\text{leftctx}(Z) = \{ \varepsilon \}$	*postulato*
$\text{leftctx}(E) \supseteq \text{leftctx}(Z) \bullet \{ \varepsilon \} = \text{leftctx}(Z)$	$Z \to E$
$\text{leftctx}(T) \supseteq \text{leftctx}(E) \bullet \{ \varepsilon \} = \text{leftctx}(E)$	$E \to \boldsymbol{T}+E$
$\text{leftctx}(E) \supseteq \text{leftctx}(E) \bullet \{ T+ \}$	$E \to T+ \boldsymbol{E}$
$\text{leftctx}(T) \supseteq \text{leftctx}(E) \bullet \{ \varepsilon \} = \text{leftctx}(E)$	$E \to \boldsymbol{T}$

Al solito, le produzioni che generano simboli terminali non si considerano.
Unendo i vari contributi:

$\text{leftctx}(Z) = \{ \varepsilon \}$	$\to$	$\langle \mathbf{LctxZ} \rangle \to \varepsilon$
$\text{leftctx}(E) = \{ \varepsilon \} \cup \text{leftctx}(E) \bullet \{ T+ \}$	$\to$	$\langle \mathbf{LctxE} \rangle \to \varepsilon \mid \langle \mathbf{LctxE} \rangle \mathbf{T+}$
$\text{leftctx}(T) = \text{leftctx}(E)$	$\to$	$\langle \mathbf{LctxT} \rangle \to \langle \mathbf{LctxE} \rangle$

$\text{leftctx}(T) = \text{leftctx}(E) = (\mathbf{T+})^*$

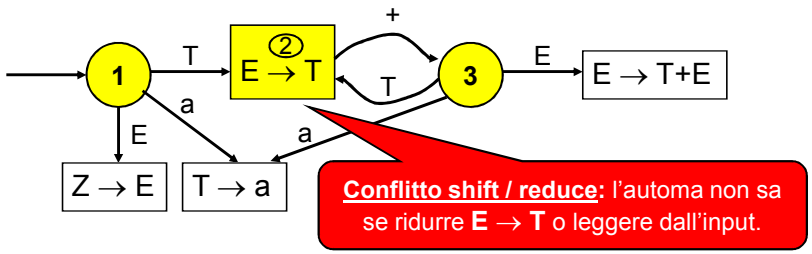
CONTROESEMPIO 1: calcolo contesti (2/3)

Si consideri la seguente grammatica:
$Z \to E$ $E \to T+E \mid T$ $T \to a$

Dunque, dal fatto che $\text{leftctx}(T) = \text{leftctx}(E) = (\mathbf{T+})^*$ segue che:

$\mathbf{ctx^{LR(0)}(Z \to E)}$	$= \text{leftctx}(Z) \bullet \{ E \}$	$= \mathbf{E}$
$\mathbf{ctx^{LR(0)}(E \to T+E)}$	$= \text{leftctx}(E) \bullet \{ T+E \}$	$= (\mathbf{T+})^* \mathbf{T+E}$
$\mathbf{ctx^{LR(0)}(E \to T)}$	$= \text{leftctx}(E) \bullet \{ T \}$	$= (\mathbf{T+})^* \mathbf{T}$
$\mathbf{ctx^{LR(0)}(T \to a)}$	$= \text{leftctx}(T) \bullet \{ a \}$	$= (\mathbf{T+})^* \mathbf{a}$

Conflitto



CONTROESEMPIO 1: calcolo contesti (3/3)

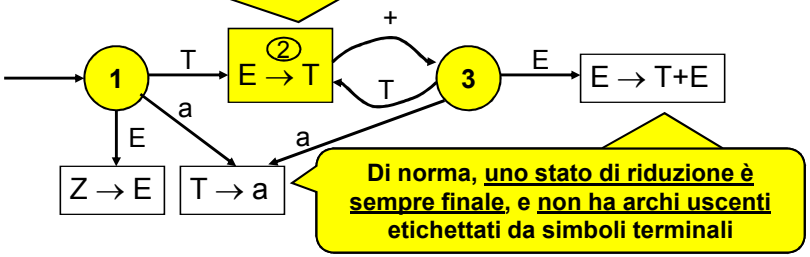
Si consideri la seguente grammatica:

$Z \to E$ $E \to T+E \mid T$ $T \to a$

Dunque, dal fatto che leftctx(T) = leftctx(E) = (T+)* segue che:

$ctx^{LR(0}$
$ctx^{LR(0}$
$ctx^{LR(0}$
$ctx^{LR(0}$

INVECE, lo stato 2 è uno stato di riduzione, ma *ha anche un arco uscente* etichettato con un simbolo terminale. *È il segno del conflitto !*



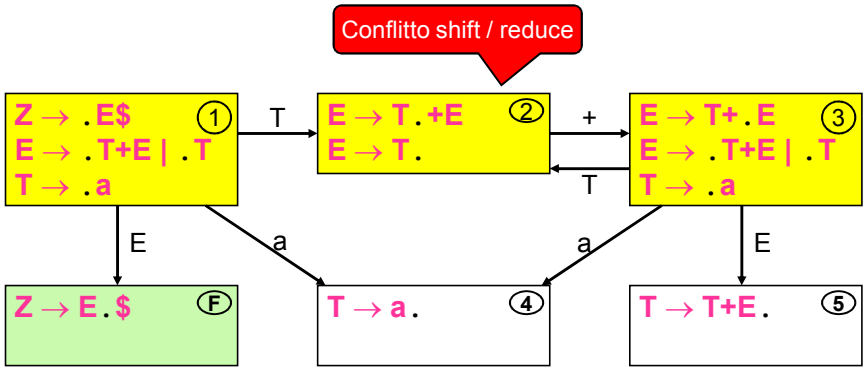
Di norma, uno stato di riduzione è sempre finale, e non ha archi uscenti etichettati da simboli terminali

CONTROESEMPIO 1: procedimento operativo

Si consideri la seguente grammatica:

$Z \to E$ $E \to T+E \mid T$ $T \to a$

In alternativa si poteva costruire l'automa col noto procedimento:



Esercizio

- Consider the following grammar G:

  S→ B

  B → (B ∧ B) | ¬B | t | f

a) Define the deterministic LR(0) parsing automaton of G.

b) Parse the input ((¬t ∧ f) ∧ f) . Provide the corresponding run of your automaton from a).

Esercizio

- Si scriva l'automa LR(0) che riconosce la grammatica

  S → AA

  A → aA | b

- Si mostri poi come viene riconosciuta la stringa **aabab**.

Si ringrazia il prof. Enrico Denti per aver
fornito la prima versione di questi lucidi