Esercizio 1

Grammatica: `S -> aSb | ε`

Costruiamo la collezione degli stati che servono all'automa caratteristico.

Stato iniziale 0:
```
S' -> .S, {$}
/* Devo chiudere la S */
- - - - - - -
S -> .aSb, {$}   Il lookahead set è dai FIRST di tutto ciò che può seguire S nell'item che mi ha costretto a inserire questi due item.
S -> ., {$} /* Corrisponde alla produzione S -> ε */
/* La chiusura non aggiunge nient'altro, in quanto non ho non-terminali proceduti dal punto */
```

Per lo stato 0 ci aspettiamo due transizioni: una per la `S`, e una per la `a`.

Dallo stato 0, con la `S`, otteniamo un kernel item che prima non avevamo collezionato.
Quindi, nuovo **stato 1**:
```
S' -> S., {$}
/* La chiusura non aggiunge niente */
```

Perchè devo preoccuparmi di controllarlo?
Perchè la chiusura di due kernel uguali non può restituire insiemi diversi! :)

Dallo stato 0, con la `a`, otteniamo un kernel item che prima non avevamo collezionato.
Quindi, nuovo **stato 2**:
```
S -> a.Sb, {$}
/* Devo chiudere la S */
- - - - - - - -
S -> .aSb, {b}   LOOKAHEAD = FIRST (b $)
S -> ., {b}
 /* La chiusura non aggiunge niente */
```

Lo stato 1 e lo stato 2 sono stati completamente processati. Tocca allo stato 2, per il quale ci aspettiamo due transizioni: una per la `S`, e una per la `a`.

Dallo stato 2, con la `S`, otteniamo un kernel item che prima non avevamo collezionato.
Quindi, nuovo **stato 3**:
```
S -> aS.b, {$}
/* La chiusura non aggiunge niente */
```

Dallo stato 2, con la `a`, otteniamo un kernel item che prima non avevamo collezionato.
Quindi, nuovo **stato 4**:
```
S -> a.Sb, {b}
/* Devo chiudere la S */
- - - - - - - -
S -> .aSb, {b}
S -> ., {b}
/* La chiusura non aggiunge niente */
```

Lo stato 2 è stato completamente processato. Tocca allo stato 3, per il quale ci aspettiamo una transizione rispetto alla `b`.

Dallo stato 3, con la `b`, otteniamo un kernel item che prima non avevamo collezionato.
Quindi, nuovo **stato 5**:
```
S -> aSb., {$}
```

Lo stato 3 è stato completamente processato. Tocca allo stato 4, per il quale ci aspettiamo due transizioni: una per la `S`, una per la `a`.

Dallo stato 4, con la `S`, otteniamo un kernel item che prima non avevamo collezionato.
Quindi, nuovo **stato 6**:
```
S -> aS.b, {b}
/* La chiusura non aggiunge niente */
```

Dallo stato 4, con la `a`, otteniamo il kernel item `S -> aS.b, {b}`, che già abbiamo nello stato 4.
Questo vuol dire che dallo stato 4, con la `a`, andiamo allo stato 4.

Lo stato 4 e lo stato 5 sono stati completamente processati. Tocca allo stato 6, per il quale ci aspettiamo una transizione rispetto alla `b`.

Dallo stato 6, con la `b`, otteniamo un kernel item che prima non avevamo collezionato.
Quindi, nuovo **stato 7**:

```
S -> aSb., {b}
/* La chiusura non aggiunge niente */
```

Lo stato 6 e lo stato 7 sono stati completamente processati.

Ho finito, passo a disegnare l'automa caratteristico. Dal momento che, successivamente, per riempire la tabella di parsing, ci servirà, oltre all'automa caratteristico, sapere anche quali sono i reducing items e i relativi lookahead set, evidenziamo nell'automa caratteristico i reducing item.

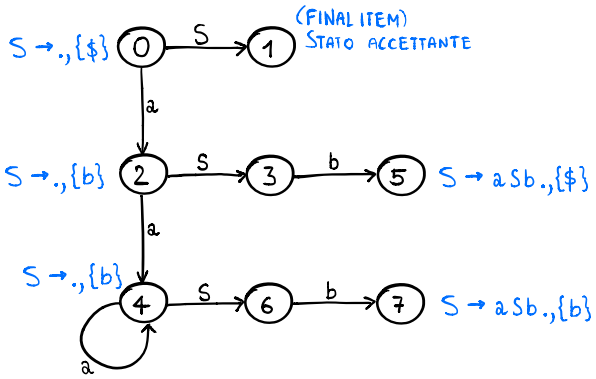
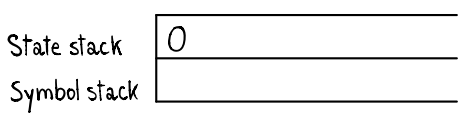


Tabella di parsing: come si riempie?

	a	b	$	S
0	S2		r "S->ε"	1
1			ACC	
2	S4	r "S->ε"		3
3		S5		
4	S4	r "S->ε"		6
5			r "S->aSb"	
6		S7		
7		r "S->aSb"		

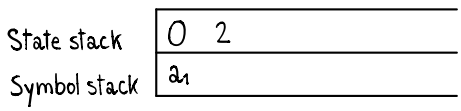
Possiamo concludere che la grammatica è LR(1) in quanto la tabella non ha entry multiply-defined.

Vediamo come si effettua il **riconoscimento** della stringa $a_1\ a_2\ b_2\ b_1$. Quando inizio, nell'automa sono nello stato 0.

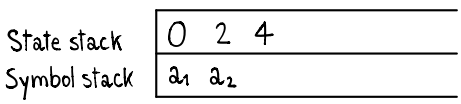


Nell'input buffer leggo a_1.

La tabella, alla entry [0, a], indica "shift 2". Metto a_1 nel symbol stack e 2 nello state stack. Nell'input buffer leggo a_2.



La tabella, alla entry [2, a], indica "shift 4". Metto a_2 nel symbol stack e 4 nello state stack. Nell'input buffer leggo b_2.



La tabella, alla entry [4, b], indica “reduce S->ε”. Tolgo un quantitativo nullo di elementi dagli stack. Metto S nel symbol stack. Nello state stack metto lo stato indicato dalla entry [4, S], cioè 6.

State stack	0 2 4 6
Symbol stack	a_1 a_2 S

La tabella, alla entry [6, b], indica “shift 7”. Metto b_2 nel symbol stack e 7 nello state stack. Nell’input buffer leggo b_1.

State stack	0 2 4 6 7
Symbol stack	a_1 a_2 S b_2

La tabella, alla entry [7, b], indica “reduce S->aSb”. Tolgo 3 elementi dagli stack. Metto S nel symbol stack.
Nello state stack metto lo stato indicato dalla entry [2, S], cioè 3.

State stack	0 2 3
Symbol stack	a_1 S

La tabella, alla entry [3, b], indica “shift 5”. Metto b_1 nel symbol stack e 5 nello state stack. Nell’input buffer leggo $.

State stack	0 2 3 5
Symbol stack	a_1 S b_1

La tabella, alla entry [5, $], indica “reduce S->aSb”. Tolgo 3 elementi dagli stack. Metto Metto S nel symbol stack.
Nello state stack metto lo stato indicato dalla entry [0, $], cioè 1.

State stack	0 1
Symbol stack	S

La tabella, alla entry [1, $], indica “accept”.

Esercizio 2

Grammatica:

```
E -> E + T | T
T -> T * id | id
```

Vogliamo stabilire se la grammatica sia analizzabile con il parsing LR canonic.

Costruiamo la collezione degli stati che servono all’automa caratteristico.