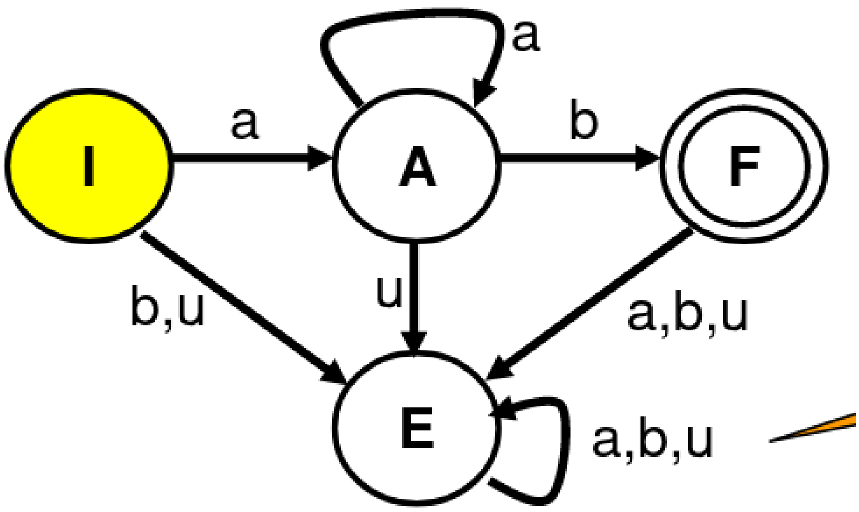
1 — Riconoscitori a stati finiti

...

1.1 Dai riconoscitori ai generatori

Figura 1.1: Automa



Descrivendo a parole le transizioni utili dell'automa:

- nello stato I l'automa può accettare:
  - il simbolo a e portarsi nello stato A
- nello stato A l'automa può accettare:
  - il simbolo b e poi fermarsi
  - il simbolo a e riportarsi nello stato A stesso
- nello stato finale F l'automa puà accettare:
  - nessun simbolo

Se si sostituisce la parola accettare alla parola generare, si nota che l'automa può essere considerato come generatore.

Si può definire un **mapping** tra:

- stati $\longleftrightarrow$ simboli non terminali
- transizioni $\longleftrightarrow$ produzioni
- scopo $\longleftrightarrow$ uno stato particolare

É possibile automatizzare la costruzione di un RSF a partire dalla grammatica o viceversa.

Grammatica regolare a destra:

- scopo = stato iniziale: I
- stato finale: F
  - $S \to a\ A$
  - $A \to a\ A \mid b$

Grammatica regolare a sinistra:

- scopo = stato finale: F
- stato inziale: I
  - $S \to A\ b$
  - $A \to A\ a \mid a$

Lo stato finale F non si considera perché non ha archi uscenti.
Come arrivare a queste grammatiche?

1.1.1 Mapping RSF $\longleftrightarrow$ grammatica

Si immagini un osservatore che, stando in ogni stato, guardi da ogni stato:

- dove si va
  - la freccia col **simbolo terminale**
  - lo stato **successivo**
- da dove viene
  - lo stato **precendente**
  - la freccia col **simbolo terminale**

Mapping tra automa riconoscitore e grammatica

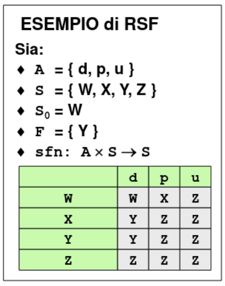
Tra la grammatica e il riconoscitori si riconoscono le seguenti corrispondenze:

- **stati** dell'automa $\longleftrightarrow$ **metasimboli** della grammatica
- **transizioni** dell'automa $\longleftrightarrow$ **produzioni** della grammatica
- **uno stato** dell'automa $\longleftrightarrow$ **scopo** della grammatica

Se la grammatica è regolare a destra si ottiene un automa top-down.
Se la grammatica è regolare a sinistra si ottiene un automa bottom-up.
In modo analogo si ottengono grammatiche destra/sinistra interpretando un RSF in modo top-down/bottom-up
Ad esempio:

In questo caso gli stati iniziale e finale sono anche stati di transito.

(a) Traduzione grammatica

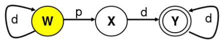

ESEMPIO di RSF

Sia:

- A = { d, p, u }
- S = { W, X, Y, Z }
- S_0 = W
- F = { Y }
- sfn: A × S → S

	d	p	u
W	W	X	Z
X	Y	Z	Z
Y	Y	Z	Z
Z	Z	Z	Z

(b) Riconoscitore



Per passare dall'automa alla grammatica si analizzano i collagamenti degli stati e si ricavano le trasformazioni:

$$\begin{array}{ll} W \to d\,W & \mid p\,X \\ X \to d & \mid d\,Y \\ Y \to d & \mid d\,Y \end{array} \qquad\qquad \begin{array}{ll} Y \to X\ d & \mid Y\ d \\ X \to p & \mid W\ p \\ W \to d & \mid W\ d \end{array}$$

1.1.2 Riconoscitori top down

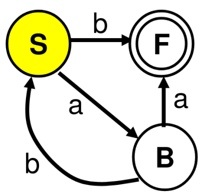
Derivazione top-down
si parte dallo scopo della grammatica e si tenta di coprire la frase data tramite produzioni successive.

Data una grammatica regolare lineare a destra, il riconoscitore:

- ha tanti **stati** quanti i simboli non terminali
- ha come **stato iniziale** lo scopo S
- per ogni regola del tipo $X \to x\ Y$, l'automa con ingresso x passa dallo stato X allo stato Y
- per ogni regola del tipo $X \to x$, l'automa con ingresso x passa dallo stato X allo stato finale F

Esempio
Sia G una grammatica lineare a destra caratterizzata dalle seguenti produzioni:

- $S \to a\ B \mid b$
- $B \to b\ S \mid a$



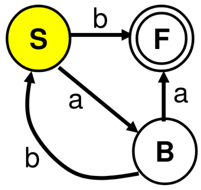
1.1.3 Riconoscitori bottom up

Data una grammatica regolare lineare a sinistra, il riconoscitore:

- ha tanti **stati** quanti i simboli non terminali
- ha come **stato finale** lo scopo S
- per ogni regola del tipo $X \to Y\ x$, l'automa con ingresso x passa dallo stato Y allo stato X
- per ogni regola del tipo $X \to x$, l'automa con ingresso x passa dallo stato iniziale I allo stato X

Esempio
Sia G una grammatica lineare a sinistra caratterizzata dalle seguenti produzioni:

- $S \rightarrow B\ a \mid b$
- $B \rightarrow S\ b \mid a$

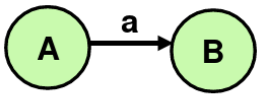


1.1.4 Dall'automa alle grammatiche

Dato un automa riconoscitore, se ne possono trarre sia una grammatica regolare a destra (interpretazione top-down) che una grammatica regolare a sinistra (interpretazione bottom-up).

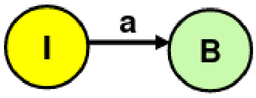
Caso generico

- top-down: $A \rightarrow a\ B$
- bottom-up: $A\ a \leftarrow B$



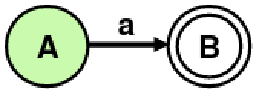
Caso iniziale

- top-down: $S \rightarrow a\ B$
- bottom-up: $I\ a \leftarrow B$

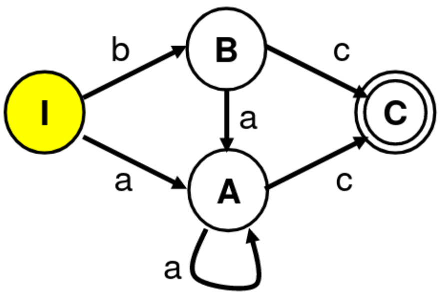


Caso finale

- top-down: $A \rightarrow a\ F$
- bottom-up: $A\ a \leftarrow S$



Esempio



Analisi top-down (C finale omesso):

- $I \rightarrow b\ B$
- $I \rightarrow a\ A$
- $B \rightarrow a\ A$
- $A \rightarrow a\ A$
- $B \rightarrow c$
- $A \rightarrow c$

Analisi bottom-up (C iniziale):

- $b \leftarrow B$
- $a \leftarrow A$
- $B\ a \leftarrow A$
- $A\ a \leftarrow A$
- $B\ c \leftarrow S$
- $A\ c \leftarrow S$

Grammatica G1 regolare a destra:

- $I \to b\ B \mid a\ A$
- $B \to a\ A \mid c$
- $A \to a\ A \mid c$

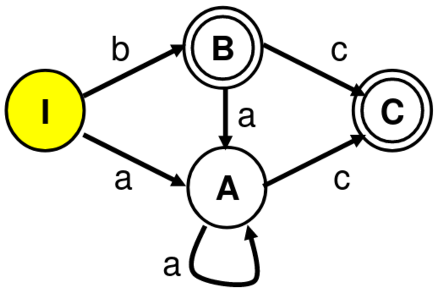
L(*G1*) =
b L(**B**) + **a** L(**A**) = (**ba**+**a**) L(**A**) + **bc** =
= (**ba**+**a**)**a*****c** + **bc** = **ba*****c** + **aa*****c** =
= (**b** + **a**) **a*** **c**

Grammatica G2 regolare a sinistra:

- $S \to B\ c \mid A\ c$
- $A \to B\ a \mid A\ a \mid a$
- $B \to b$

L(*G2*) =
L(**B**) **c** + L(**A**) **c** = (**b** + L(**A**)) **c** =
(**b** + (**b a** + **a**) **a***) **c** = (**b** + **a**) **a*** **c**
perché (b + b a a) = b a**

Esempio multipli stati finali



Analisi top-down (regola in più sulla I):

- $I \to b\ B \mid b$
- $I \to a\ A$
- $B \to a\ A$
- $A \to a\ A$
- $B \to c$
- $A \to c$

Analisi bottom-up (C iniziale):

- $b \leftarrow B$
- $a \leftarrow A$
- $B\ a \leftarrow A$
- $A\ a \leftarrow A$
- $B\ c \leftarrow C$
- $A\ c \leftarrow C$

L'analisi bottom-up deve scegliere quale scopo adottare, se B o C. Per risolvere questo problema si creano due grammatiche in cui si assumono a turno $C \equiv S$ e $B \equiv S$.

*Grammatica G2' (assume **C ≡ S**):*
S → B c | A c
A → B a | A a | a
B → b
L(*G2'*) = (**b** + **a**) **a*** **c**

*Grammatica G2'' (assume **B ≡ S**):*
b ← S
le altre regole sono inutili, essendo i loro metasimboli irraggiungibili !
L(*G2''*) = **b**

Caso generale

Nel caso bottom-up, in presenza di più stati finali:

- si assume come (sotto)scopo S_k uno stato finale per volta
- si scrivono le regole bottom-up corrispondenti
- si esprime il linguaggio complessivo come unione dei vari sotto-linguaggi, definendo lo scopo globale come $S \to S_1 \mid S_2 \mid \cdots \mid S_n$

2 — Riconoscitori con PDA

Un RSF non può riconoscere un linguaggio di tipo 2, ha un limite intrinseco alla capacità di memorizzazione: non riesce a riconoscere frasi che richiedano di memorizzare una parte di lunghezza non nota a priori.

Esempio: bilanciamento delle parentesi $L = (^n c)^n, n \geq 0$, $G = S \rightarrow (S) \mid c$

In questo linguaggio il prefisso $(^n$ non ha lunghezza limitabile a priori.

2.1 Push-Down Automaton (PDA)

Un PDA è un RSF con aggiunto uno stack, con stack non ci si riferisce a una struttura dati fisica ma a un suo modello astratto, ovvero una sequenza di simboli, definito in modo tale che si possa operare soltanto su quello in "cima".

Il PDA legge un simbolo d'ingresso e transita in un nuovo stato, in più a ogni passo altera lo stack, producendo una nuova configurazione.

Un PDA può prevedere ε-mosse, ovvero transizioni spontanee che manipolano lo stack senza consumare simboli in ingresso.

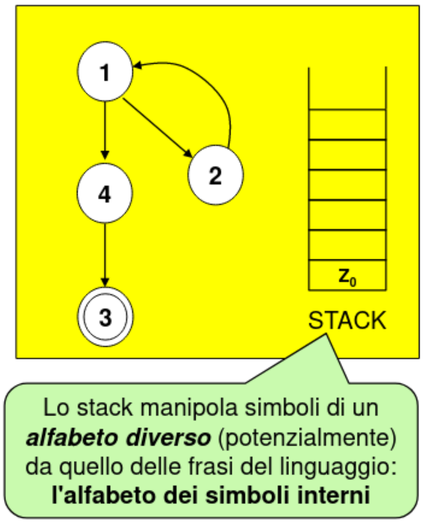
Un PDA è una sestupla:

$$\langle A,\ S,\ S_0,\ sfn,\ Z,\ Z_0 \rangle$$

dove:

- A = alfabeto
- S = insieme degli stati
- S_0 = stato iniziale $\in S$
- $sfn : (A \cup \varepsilon) \times S \times Z \rightarrow W$
- Z = alfabeto dei simboli interni
- $Z_0 \in Z$ = simbolo iniziale sullo stack

Figura 2.1: Struttura PDA



Il linguaggio accettatto da un PDA è definibile in 2 modi equivalenti:

- **Criterio dello stato finale**: il linguaggio accettato è l'insieme di tutte le stringhe di ingtesso che portano il PDA in uno degli stati finali.
- **Criterio dello stack vuoto**: il linguaggio accettato è definito come l'insieme di tutte le stringhe di ingresso che portano il PDA nella configurazione di *stack vuoto*.

La funzione sfn, dati:

- un *simbolo in ingresso* $a \in A$
- lo *stato attuale* $s \in S$
- il simbolo interno attualmente al *top dello stack* $z \in Z$

opera come segue:

- **consuma** il simbolo di ingresso a
- effettua il **POP** dello dallo stack, prelevando il simbolo z
- porta l'automa in stato futuro $s' = sfn(a, s, z)_{\prod S}$
- effettua una **PUSH** sullo stack di zero o più simboli interni $z' \in Z$ $z' = sfn(a, s, z)_{\prod S}$

Si consideri il linguaggio generato da:

- $A = \{0, 1, c\}$
- $P = \{S \to 0\ S\ 0 \mid 1\ S\ 1 \mid c\}$
- $L = \{\texttt{word}\ c\ \texttt{word}^R\}$

Definiamo il PDA come segue:

- $A = \{0, 1, c\}$
- $S = \{Q1 = S_0, Q2\}$
- $Z = \{Zero, Uno, Centro\}$

A ∪ ε	S	Z	S × Z*
0	Q1	Centro	Q1 × CentroZero
1	Q1	Centro	Q1 × CentroUno
c	Q1	Centro	Q2 × Centro
0	Q1	Zero	Q1 × ZeroZero
1	Q1	Zero	Q1 × ZeroUno
c	Q1	Zero	Q2 × Zero
0	Q1	Uno	Q1 × UnoZero
1	Q1	Uno	Q1 × UnoUno
c	Q1	Uno	Q2 × Uno
0	Q2	Zero	Q2 × ε
1	Q2	Uno	Q2 × ε
ε	Q2	Centro	Q2 × ε

2.1.1 PDA non deterministici

Anche un PDA può essere non deterministico: in tal caso, la funzione sfn produce insiemi di elementi W (W sottoinsieme finito di $S \times Z^*$).

Ad esempio, il PDA tale che $sfn(Q_0, a, Z) = \{(Q_1, Z_1)(\cdots)(Q_k, Z_k)\}$ è **non deterministico** in quanto l'automa, nello stato Q_0, con simbolo interno in cima allo stack Z e ingresso a ha più di uno stato futuro possibile in base alla evoluzione cambia anche il se di simboli da porre nello stack.

TEOREMA

La classe dei linguaggi riconosciuti da un PDA non-determistico coincide con la classe dei linguaggi context-free: perciò qualunque linguaggio context-free può sempre essere riconosciuto da un opportuno PDA.

É possibile rinunciare al determinismo?

In generale no:

TEOREMA

Esistono linguaggi context-free riconoscibili soltanto da PDA non-deterministici.

ma in molti casi di interesse pratico esistono linguaggi context-free riconoscibili da PDA deterministici: linguaggi context-free deterministici.

2.1.2 PDA deterministici

Cosa serve per ottenerlo?

Viste le condizioni precedenti, non deve succedere che l'automa, in dato stato Q_0, con simbolo in cima allo stack Z e ingresso x possa:

- portarsi in più stati futuri $sfn(Q_i, x, Z) = \{(Q_1, Z_1), (\cdots), (Q_k, Z_k)\}$
- optare se leggere o non leggere il simbolo di ingresso x a causa della presenza di entrambe le mosse $sfn(Q_i, x, Z)$ e $sfn(Q_i, \varepsilon, Z)$

Unendo, intersecando o concatenando linguaggi deterministici, non necessariamente si ottiene un linguaggio deterministico.

I complemento di un linguaggio è deterministico.

Con L linguaggio deterministico e R linguaggio regolare, il linguaggio quoziente L/R (insieme di stringhe di L private di suffisso regolare) è deterministico.

Con L linguaggio deterministico e R linguaggio regolare, il concatenamento $L.R$ (insieme di stringhe di L cun suffisso regolare) è deterministico.

Sottoclassi particolari

Per un PDS **deterministico**:

- il criterio dello stack vuoto risulta meno potente del criterio stati finali
- una limitazione sul numero di stati intenro o sul numero di configurazioni finali riduce l'insieme dei linguaggi riconoscibili
- l'assenza di ε-mosse riduce l'insieme dei linguaggi riconoscibili

2.1.3 Realizzazione di PDA deterministici

Possibilità di seguire la definzione, ma non molto pratico.

Si adotta un approccio che manipoli uno stack con la stessa logica di un PDA, dove lo stack è la vera differenza, ad esempio una macchina virtuale che abbia uno stack può essere fatta funzionare come PDA, opportunamente pilotata.

Si potrebbe controllare "a mano" lo stack, oppure in modo automatico attraverso le chiamate ricorsive di funzioni, dove sono già gestiti stack relativi alla ricorsione.

Top-Down Recursive-Descent Parsing

Con l'**analisi ricorsiva discendente** si introduce una funzione per ogni metasimbolo della grammatica, e la si chiama ogni volta che si icontra quel metasimbolo.

Ogni funzione copre le regole di quel metasimbolo, ossia riconosce il sottolinguaggio corrispondente:

- termina normalmente (o segno di successo) se incontra simboli coerenti
- abortisce (o restituisce un segno di fallimento) se incontra simboli non coerenti

Esempio

Il solito linguaggio $L = \{\texttt{word}\ c\ \texttt{word}^n\}$, alfabeto $A = \{0,\ 1,\ c\}$ e regole $S \rightarrow 0\ S\ 0 \mid 1\ S\ 1 \mid c$.

- Introdurre tante funzioni quanti i metasimboli, qui una sola $S()$.
- Chiamare una funzione ogni volta che si incontra il suo metasimbolo
- Ogni funzione deve coprire le regole di quel metasimbolo
  - se il simbolo d'ingresso è 0 $\rightarrow$ seguire la prima regola
  - se il simbolo d'ingresso è 1 $\rightarrow$ seguire la seconda regola

– se il simbolo d'ingresso è c → seguire la terza regola

Nel caso della prima regola, consumiamo il carattere di ingresso 0, invochiamo la funzione $S()$ e consumiamo un nuovo carattere d'ingresso e verifichiamo che sia 0.

Se la verifica ha esito positivo, significa che la funzone ha incontrato simboli coerenti con le proprie regole.

Se la verifica ha esito negativo, significa che la funzione ha incontrato simboli che non corrispondono alle sue regole.

finisci slide esempio 24-28!!!

2.1.4 Separare motore e grammatica

Applicare l'analisi ricorsiva discendente è un processo meccanico, che tuttavia introduce informazioni cablate nel codice, difficili da aggiornare.

É possibile separare il **motore**, invariante rispetto alle regole, dalle regole della grammatica; si presta a questo scopo la tabella di parsing, simile alla tabella delle transizioni di un RSF, indica però la prossima *produzione* da applicare.

Il motore prenderà singole decisioni consultando questa tabella.

Parsing tables - esempi deterministici

Linguaggio $L = \{\texttt{word}\ c\ \texttt{word}^n\}$

	0	1	c
S	S → 0 S 0	S → 1 S 1	S → c

Linguaggio $L = \{$`if c then cmd` (`endif` | `else cmd`) $\}$

Produzioni $S \rightarrow$ `if c then cmd X`, `X` $\rightarrow$ `endif` | `else cmd`

	if	c	then	endif	else	cmd
S	S → if c then cmd X	*error*	*error*	*error*	*error*	*error*
X	*error*	*error*	*error*	X → endif	X → else cmd	*error*

2.1.5 Analisi ricorsiva discendente - vantaggi e limiti

Vantaggi

- immediata scrittura del riconoscitore
- migliore leggibilità a modificabilità del codice
- facilitata inserzione di azioni nella fase di analisi

Svantaggi

- non sempre applicabile

- funzionale solo se non esistono ambiguità sulla regola da applicare

Si individua una sottoclasse di grammatiche context-free che garantisce il determinismo dell'analisi sintattica.

Per rendere deterministica l'analisi ricorsiva, si rende necessario avere una visione del *passato* dell'analisi (simboli consumati fino a quel punto) e una del *futuro*, generalmente un solo carattere in avanti.

2.2 Grammatiche $LL(k)$

Si definiscono *grammatica* $LL(k)$ quelle che sono analizzabili in modo deterministico:

- procedendo *left to right*
- applicando *left-most derivation*
- guardando avanti al più k simboli

Ricoprono un posto fondamentale le grammatiche LL(1), ovvero quelle in cui basta guardare un simbolo in avanti per operare in modo deterministico.

Esempio

Si consideri la grammatica

- `VT` $=$ {`p`, `q`, `a`, `b`, `d`, `x`, `y`}
- `VN` $=$ {`S`, `X`, `Y`}

Produzioni

$$\begin{array}{ll} \mathtt{S} \rightarrow \mathtt{p\ X} & |\ \mathtt{q\ Y} \\ \mathtt{X} \rightarrow \mathtt{a\ X\ b} & |\ \mathtt{x} \\ \mathtt{Y} \rightarrow \mathtt{a\ Y\ d} & |\ \mathtt{y} \end{array}$$

Le parti **destre** delle produzioni di uno stesso meta simbolo, iniziano tutte con un simbolo terminale diverso, è sufficiente guardare avanti di un carattere per scegliere con certezza la produzione per scegliere con certezza la produzione con cui proseguire l'analisi.

Creando la parsing table, si nota che ogni cella contiene una sola produzione, quindi non si hanno ambiguità sulle prossime mosse da fare, è facile vedere che il parser è deterinistico.

	p	q	a	b	d	x	y
S	S → p X	S → q Y	error	error	error	error	error
X	error	error	X → a X b	error	error	X → x	error
Y	error	error	Y → a Y d	error	error	error	Y → y

La frase di inzio deve essere completa oppure può essere parziale? Nel caso in cui si voglia imporre che la frase deve essere finale occorre imporre una regola *top-level* che specifica che la frase deve terminare con `$`, carattere che rappresenta il fine stringa/linea/file del tipo `Z` $\rightarrow$ `S $` .

2.2.1 Starter Symbol Set

Spesso le parti destre delle produzioni di uno stesso metasimbolo non iniziano tutte con un simbolo terminale, non è chiaro quali siano gli input ammissibili.

Occorre ridefinire il concetto di simbolo iniziale → Starter Symbols Set.

Lo starter symbols set della riscrittura α è l'insieme

$$SS(\alpha) = \{a \in \mathtt{VT} \mid \alpha \rightarrow^* a\ \beta\},\ \ \mathtt{con}\ \ \alpha \in V^+\ \ \mathtt{e}\ \ \beta \in V^*$$

In sostanza gli starter symbols sono le iniziali di una forma di frase α, ricavate applicando con più produzioni. L'operatore $*$ cattura il caso limite in cui $\alpha \in VT$, ossia già terminale, e non richiede di applicare derivazioni.

Per includere anche il caso $\alpha \rightarrow \varepsilon$, si introduce l'insieme

$$\mathtt{FIRST}(\alpha) = \mathtt{trunc}_1(\{x \in VT^* \mid\ \rightarrow^*\}),\ \ \mathtt{con}\ \ \alpha \in V^*$$

dove $\mathtt{trunc}_1$ denota il troncamento della stringa al primo elemento.

Generalizzando la regola precendete, condizione necessaria perché una grammatica sia $LL(1)$ è che gli *start symbols* relativi alle parti destre di uno stesso metasimbolo siano disgiunti.

Esempi 44-52???

É possibile capire in modo più rapido se una grammatica è $LL(1)$?

Due opzioni:

- agire sulla parsing table, formalizzando il concetto di **blocco annullabile** e integrando nella tabella l'informazione sulle stringhe che possono scomparire.
- ampliare la nozione di starte symbols set con i Director Symbols set (o Look-Ahed set)

inserisci slide 55-67!!!1