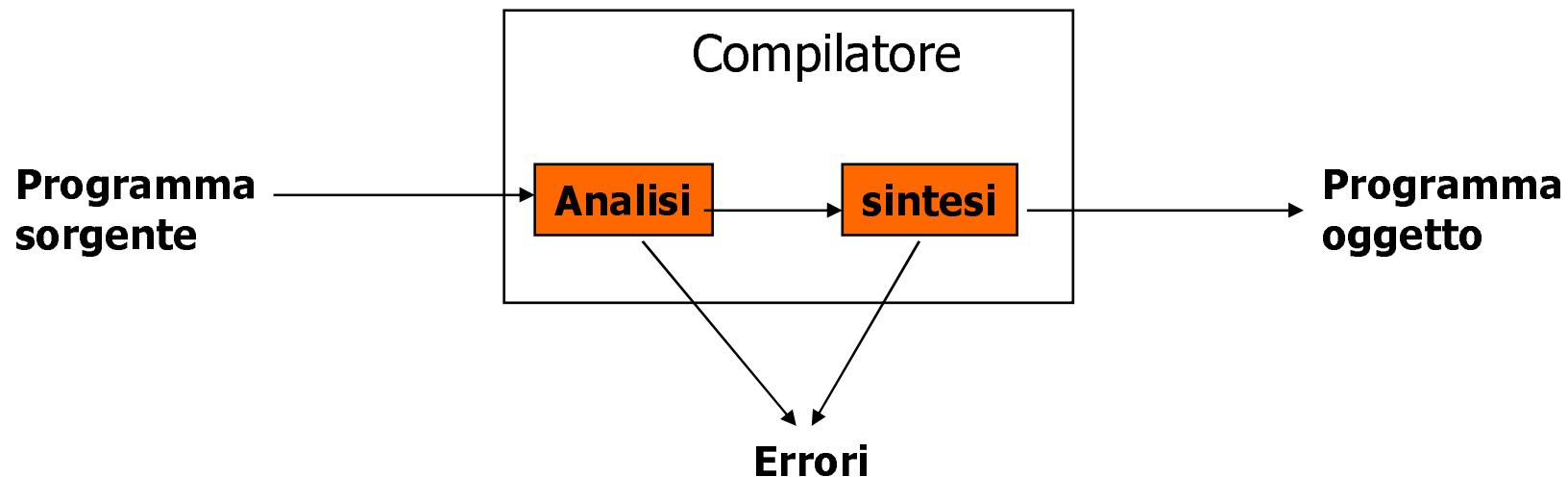


# **Linguaggi Formali e Compilatori**

# I compilatori

⌘ Un **compilatore** è un **traduttore** in grado di trasformare un programma scritto in un linguaggio (**sorgente**) in un programma scritto in un altro linguaggio (**oggetto**)



# I compilatori II



⌘ Il processo di compilazione consiste di due parti: **analisi** e **sintesi**

⌘ analisi

- ⏏ divide il programma sorgente nei suoi componenti elementari

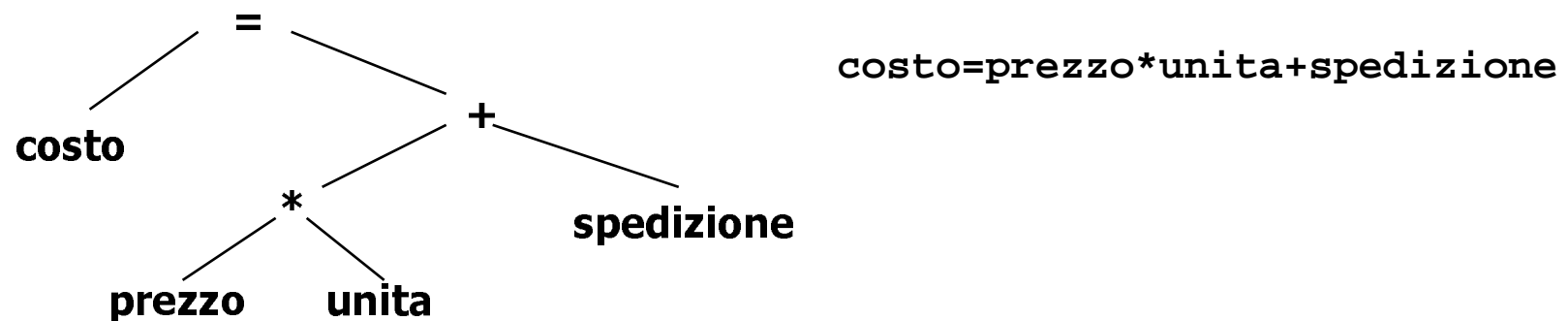
- ⏏ basandosi sulle relazioni fra i componenti crea una **rappresentazione intermedia** del programma

⌘ sintesi

- ⏏ costruisce il programma oggetto dalla rappresentazione intermedia

# Analisi

- ⌘ I componenti elementari di un programma sono: identificatori, operatori, variabili,...
- ⌘ La rappresentazione intermedia è codificata da una struttura dati a forma di albero (**albero sintattico**)



# Sintesi



- ⌘ L'albero sintattico descrive il programma sorgente e le relazioni fra i suoi componenti elementari
- ⌘ La fase di sintesi usa questa informazione per produrre un codice oggetto **semanticamente equivalente** al codice sorgente
- ⌘ Il codice oggetto è un programma che può essere eseguito sulla stessa macchina o su una diversa

# Campi di applicazione



⌘ Molti strumenti implementano tecniche derivanti da quelle impiegate nei compilatori

- ⌘ Interpreti

- ⌘ Editori guidati da sintassi

- ⌘ RAD

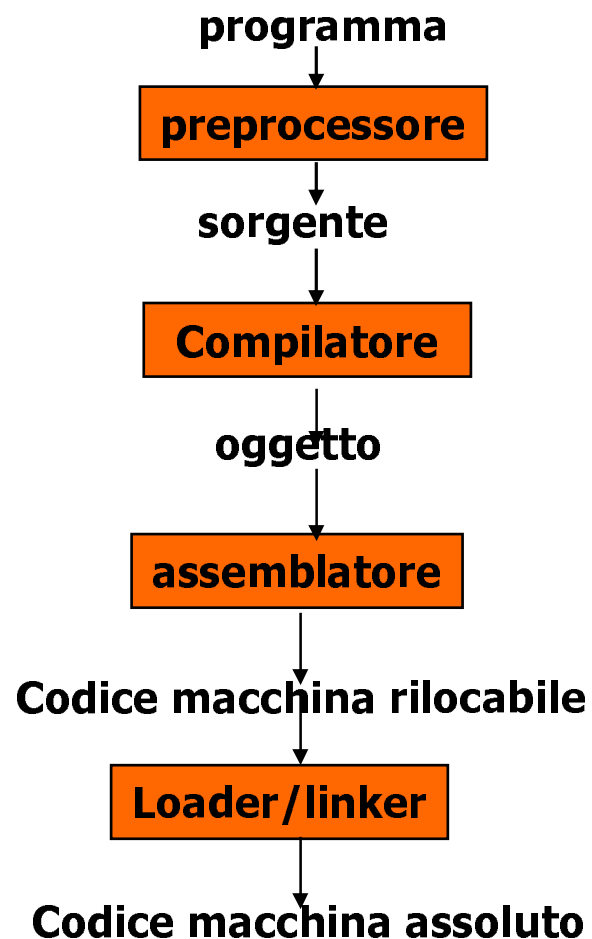
- ⌘ Browser

- ⌘ Word processor

- ⌘ Formattatori di testo

- ⌘ Interpreti di interrogazioni

# L'uso di un tipico compilatore



⌘ La tipica compilazione di un programma prevede l'uso di altri strumenti oltre al compilatore

⌘ un preprocessore

⌘ un assemblatore

⌘ un loader/linker

# La struttura di un compilatore

⌘ Un compilatore opera in diverse fasi

⌘ **analisi lessicale:**

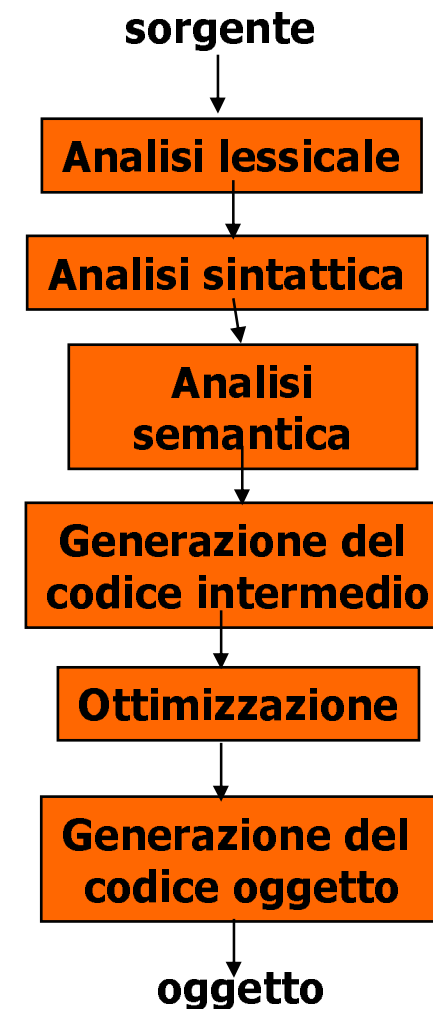
individua le componenti elementari

⌘ **analisi sintattica:**

generazione dell'albero sintattico

⌘ **analisi semantica:**

si fanno verifiche sul programma in ingresso per accertarsi che sia corretto (ad es., una variabile è dichiarata prima di essere usata)





# La struttura di un compilatore II

## ⚡ generazione codice intermedio:

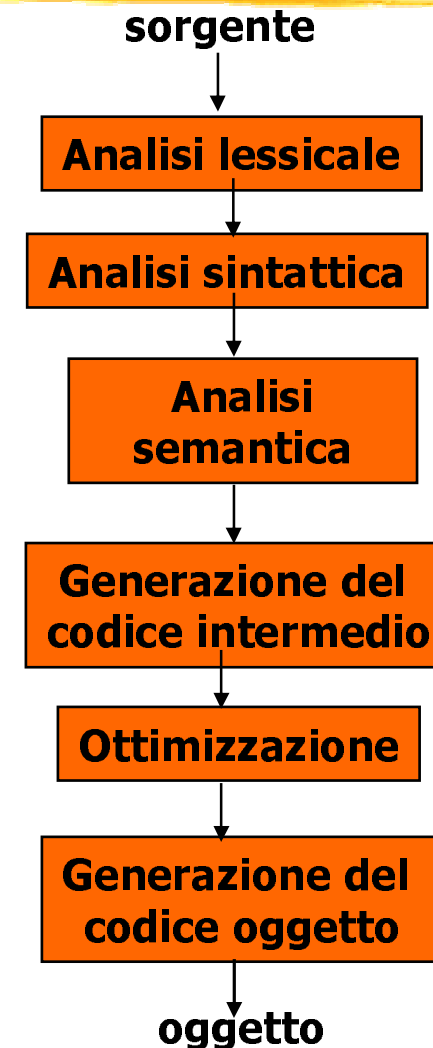
si genera un codice intermedio indipendente dalla macchina per la quale si compila

## ⚡ ottimizzazione:

il codice viene ottimizzato (ad es., si rimuove il codice non raggiungibile)

## ⚡ generazione del codice oggetto:

Il codice intermedio ottimizzato viene tradotto nel codice oggetto



# Analisi lessicale e sintattica

## ⌘ Analisi lessicale

`costo=prezzo*10+spedizione`



`costo=unità*10+spedizione`

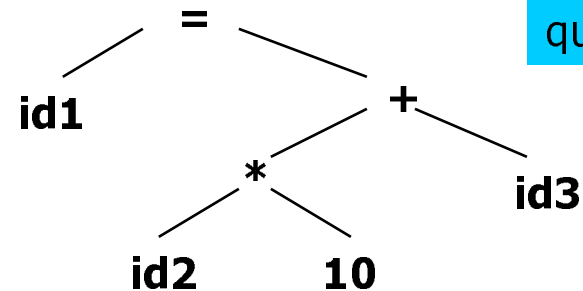
Tabella dei simboli

<b>costo</b>	....
<b>unita</b>	...
<b>spedizione</b>	....
.....	....

E' inizializzata  
dall'analizzatore  
lessicale e  
completata da  
quello sintattico

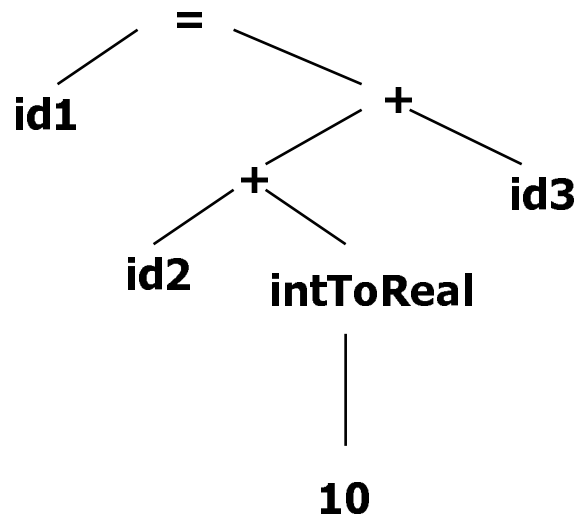
## ⌘ Analisi sintattica

`id1=id2*10+id3`



# Analisi semantica

## ⌘ Analisi semantica

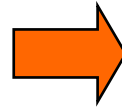
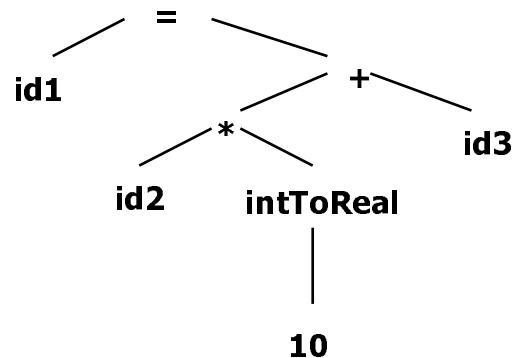


### Controlli

- `id1, id2, id3` sono stati definiti ?
- `id2, id3` sono stati inizializzati?
- `id1, id2, id3` sono di tipo Real?
- .....

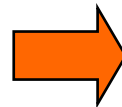
# Generazione del codice intermedio e ottimizzazione

## ⌘ Generazione codice intermedio



```
tmp1=intToReal(10)
tmp2=id2*tmp1
tmp3=id3+tmp2
id1=tmp3
```

## ⌘ Ottimizzazione



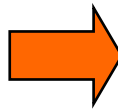
```
tmp1=id2*10.0
id1=tmp1+id3
```

# Generazione del codice oggetto

## ⌘ Generazione codice oggetto

```
tmp1=id2*10.0  
id1=tmp1+id3
```

**Codice generico**



```
movf id2, R2  
mulf #10.0, R2  
movf id3, R1  
addf R2, R1  
movf R1, id1
```

**Assembler per una  
architettura specifica**

# Cosa studieremo



⌘ Ci concentreremo sull'analisi lessicale e sintattica (le prime due fasi)

⌘ analisi lessicale:

- ⊞ linguaggi regolari (i linguaggi che riconosce)

- ⊞ come si costruisce un analizzatore lessicale

⌘ analisi sintattica:

- ⊞ linguaggi liberi da contesto (i linguaggi che riconosce)

- ⊞ esistono vari tipi di analizzatori sintattici, ne vedremo alcuni

# **Analisi lessicale**

# Linguaggi



- ⌘ Un **alfabeto**  $\Sigma = \{a_1, a_2, \dots, a_n\}$  è un insieme di simboli
- ⌘ Una **stringa** è una sequenza finita di simboli
- ⌘ Con l'insieme  $\Sigma^*$  si rappresenta l'insieme di tutte le stringhe sull'alfabeto  $\Sigma$
- ⌘ Un **linguaggio**  $L(\Sigma)$  sull'alfabeto  $\Sigma$  è un sottoinsieme di  $\Sigma^*$ , cioè  $L(\Sigma) \subseteq \Sigma^*$



# Linguaggi II

⌘ Ad esempio, l'insieme dei numeri binari reali

$$\boxed{\wedge} \Sigma = \{0, 1, ". "\}$$

$$\boxed{\wedge} \Sigma^* = \{0, 1, 0.0, 0.1, \dots, \boxed{0.0.0, 0.0}, 1, \dots\}$$

contiene stringhe che **non rappresentano numeri reali**

$$\boxed{\wedge} L(\Sigma) = \{0, 1, 0.0, 0.1, \dots, \dots\}$$

contiene solo i numeri reali corretti

# Linguaggi III



⌘ Un linguaggio  $L(\Sigma)$  può essere definito

- ⊞ elencando le stringhe (se è finito)

- ⊞ definendo delle regole che

  - ⊞ verificano se una stringa appartiene a  $L$

  - ⊞ costruiscono tutte le stringhe che appartengono a  $L$

⌘ Le **espressioni regolari** definiscono una classe di **linguaggi regolari**

# Espressioni regolari

⌘ Le **espressioni regolari** definiscono un'algebra analoga a quella delle espressioni aritmetiche

⌘ Simboli atomici

- ⊞ Un carattere **c** ( $c \in \Sigma$ )
- ⊞ Il simbolo  **$\epsilon$**  (la stringa vuota)
- ⊞ Il simbolo  **$\emptyset$**  (insieme vuoto)
- ⊞ Una variabile

⌘ 3 operatori

- ⊞ La concantenazione (binario)
- ⊞ L'unione **|** (binario)
- ⊞ La chiusura di Kleene **\*** (unario)

Un'espressione regolare  
sull'alfabeto  $\Sigma = \{0, 1\}$

**0(0 | 1)\***

Le parentesi  
definiscono la  
precedenza fra gli  
operatori

# Il valore delle espressioni regolari

⌘ Il **valore di un'espressione** regolare **E** è dato dal linguaggio che essa rappresenta **L(E)**

⌘ Valore dei simboli atomici

☒  $L(c) = \{c\}$  ( $c$  è un simbolo dell'alfabeto  $\Sigma$ )

☒  $L(\varepsilon) = \{\varepsilon\}$  ( $\varepsilon$  è un simbolo speciale che rappresenta la stringa vuoto,  $\varepsilon \notin \Sigma$ )

☒  $L(\emptyset) = \emptyset$  (l'insieme vuoto)

☒ Per una variabile  $v$  a cui è stata associata un'espressione  $E$ ,  
 $L(v) = L(E)$

# L'unione

⌘ Il valore dell'operatore di **unione** è definito da

⊞ Se R e S sono due espressioni regolari, allora

$$L(R \mid S) = L(R) \cup L(S)$$

⊞ ad esempio, l'espressione che rappresenta il linguaggio che contiene 1, 0 e la stringa vuota è

$$L(1 \mid 0 \mid \epsilon)$$

⌘ L'operatore  $\mid$  è **associativo** (l'unione fra insiemi è associativa)

$$\boxplus R \mid S \mid T = (R \mid S) \mid T = R \mid (S \mid T)$$

# La concatenazione

- ⌘ La **concatenazione** non è rappresentata da alcun simbolo, ma dalla giustapposizione di due espressioni RS
- ⌘ Il valore della concatenazione RS è costruito concatenando tutte le stringhe in R con quelle in s

$$\boxtimes L(RS) = \{rs \mid r \in R, s \in S\}$$

Concatenazione fra stringhe

- ⌘ La concatenazione è associativa

$$\boxtimes RST = (RS)T = R(ST)$$

# La concatenazione II

## ⌘ Esempio 1

$$\boxplus R=\{+,-\}, S=\{0,1\}$$

$$\boxplus L(RS)=\{+0,-0,+1,-1\}$$

## ⌘ Esempio 2

$$\boxplus R=\{\text{anda,da}\}, M=\{\text{re,to}\}$$

$$\boxplus L(RS)=\{\text{andare,dare,andato,dato}\}$$

# La cardinalità

⌘ Con  $|L(E)|$  si denota la **cardinalità** di  $L$ , ovvero il numero delle stringhe contenute in  $E$

⌘ La cardinalità soddisfa le seguenti proprietà

$$\boxplus |L(R|S)| \leq |L(R)| + |L(S)|$$

$$\boxplus |L(RS)| \leq |L(R)| |L(S)|$$

⌘ E' possibile sostituire  $\leq$  con  $=$  ?



# La chiusura

⌘ L'operatore di **chiusura** di Kleene  $*$  indica 0 o più ripetizioni

⌘  $L(R^*)$  contiene

☒ La stringa vuota  $\varepsilon$

☒ tutte le stringhe di  $L(R)$ ,  $L(RR)$ ,  $L(RRR)$ ,  $L(RRRR)$ ,...

⌘ Esempio

☒  $R = \{0,1\}$

☒  $L(R^*)$  è l'insieme di tutti gli interi binari più la stringa vuota

☒  $L(R^*) = \{\varepsilon, 0, 1, 00, 01, 10, 11, \dots\}$

# La chiusura

⌘ Nel seguito  $R^n$  rappresenta  $R$  ripetuto  $n$  volte

$$\boxplus R^0 = \{\varepsilon\}, R^1 = R, R^2 = RR, R^3 = RRR$$

⌘ Formalmente:

$$\boxplus R^0 = \{\varepsilon\}, R^n = R R^{n-1}$$

$$\boxplus L(R^*) = \bigcup_{n=0..∞} L(R^n)$$

⌘ Attenzione:  $L(R^*)$  non contiene stringhe infinite !!

# Esempi di espressioni regolari

## ⌘ Gli identificatori in PASCAL

**Carattere (Carattere | Cifra)\***

⌘ dove si è definito

**Carattere = A | B ... | Z | a | b | ... | z**

**Cifra = 0 | 1 | ... | 9**

## ⌘ La **precedenza fra gli operatori** è la seguente

⌘ chiusura

⌘ concatenazione

⌘ unione

# Esempi di espressioni regolari II

⌘ Tutte le stringhe di 0 e 1 che terminano con 0

**$(0|1)^*0$**

⌘ Tutte le stringhe di 0 e 1 con almeno un 1

**$(0|1)^*1(0|1)^*$**

⌘ Tutte le stringhe di 0 e 1 con al più un 1

**$0^*10^*$**

⌘ Tutte le stringhe di 0 e 1 tali che la terza posizione a partire da destra è 1

**$(0|1)^*1(0|1)(0|1)$**

# Esempi di espressioni regolari

## III

⌘ Tutte le stringhe di 0 e 1 che contengono un numero pari di 1

**$(0 | 10^*1)^*$**

⌘ Tutte le stringhe di 0 e 1 tali che le sequenze di 1 hanno lunghezza pari

**$(0 | 11)^*$**

⌘ Tutte le stringhe di 0 e 1 che rappresentano multipli di 3

**$(0 | 1(01^*0)^*1)^*$**

# Proprietà algebriche delle espressioni regolari

⌘ Due espressioni regolari  $E$ ,  $S$  si dicono equivalenti  $E \equiv S$  se

$$L(R) = L(S)$$

⌘ Le proprietà algebriche delle espressioni regolari ricordano quelle delle espressioni aritmetiche (concatenazione = prodotto, unione=somma), ma la **concatenazione non è commutativa**

⌘ Elemento neutro

⊠  $E\varepsilon \equiv \varepsilon E \equiv E$  ( $\varepsilon$  è l'elemento neutro della concatenazione)

⊠  $E|\varnothing \equiv \varnothing|E \equiv E$  ( $\varnothing$  è l'elemento neutro dell'unione)

⌘ Elemento nullo

⊠  $E\varnothing \equiv \varnothing E \equiv \varnothing$  ( $\varnothing$  è l'elemento nullo della concatenazione)

# Proprietà algebriche delle espressioni regolari II

## ⌘ Commutatività

$$\boxed{\wedge} R|S \equiv S|R$$

$\boxed{\wedge}$  (la concatenazione non è commutativa)

## ⌘ Associatività

$$\boxed{\wedge} R | S | T \equiv (R | S) | T \equiv R | (S | T)$$

$$\boxed{\wedge} RST \equiv (RS)T \equiv R(ST)$$

## ⌘ Distributività della concatenazione sull'unione

$$\boxed{\wedge} (S|T)R \equiv (SR|TR)$$

$$\boxed{\wedge} R(S|T) \equiv (RS|RT)$$

# Proprietà algebriche delle espressioni regolari III

⌘ Idempotenza dell'unione

$$\boxed{\wedge} R|R \equiv R$$

⌘ Proprietà della chiusura

$$\boxed{\wedge} \emptyset^* \equiv \varepsilon$$

$$\boxed{\wedge} R^*R \equiv RR^*$$

$$\boxed{\wedge} R^*R \mid \varepsilon \equiv R^*$$

⌘ Esempio

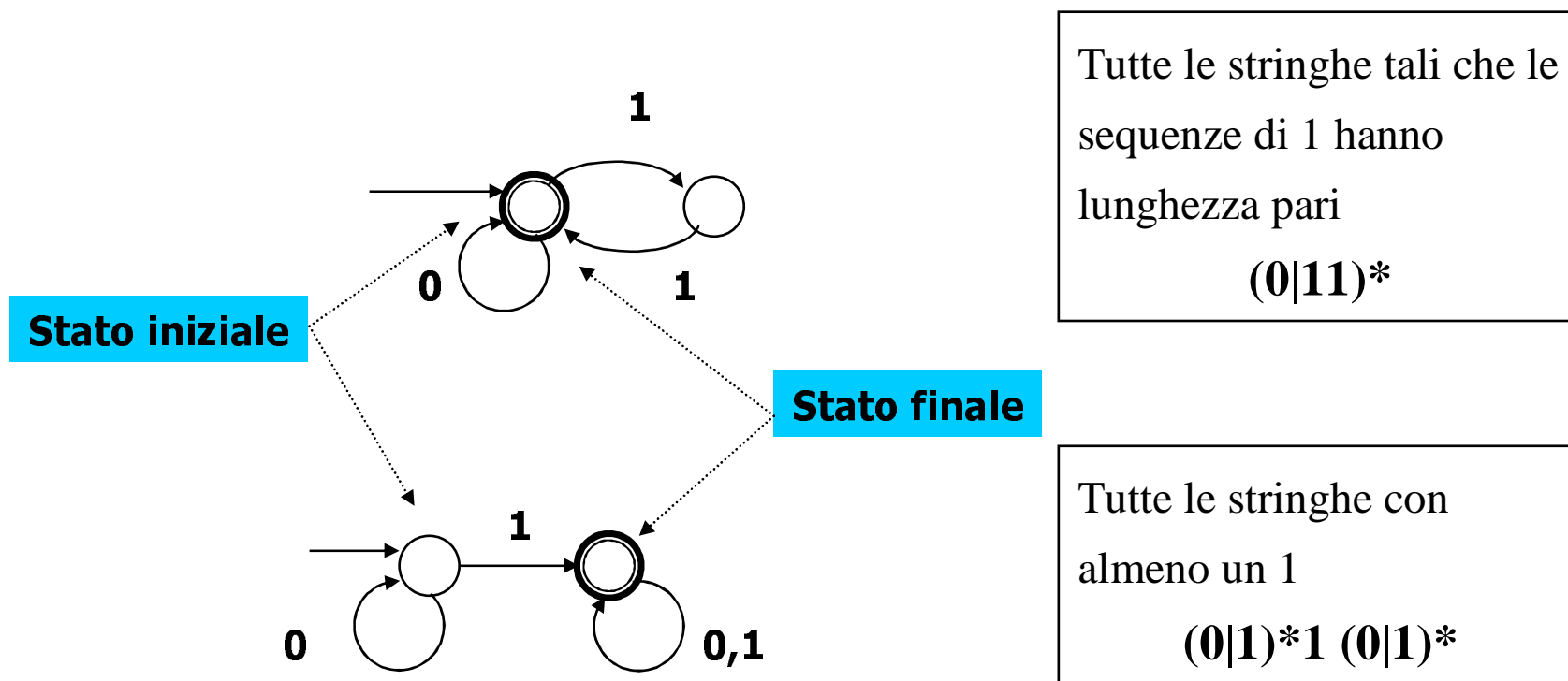
$$\boxed{\wedge} (0|1)^*(10|11)(0|1)$$

$$\boxed{\wedge} \equiv (0|1)^*1(0|1)(0|1)$$

$$\boxed{\wedge} \equiv (0|1)^*1(00|01|10|11)$$



# Espressioni regolari e automi a stati finiti



# Espressioni regolari e automi a stati finiti II

⌘ Si può dimostrare che

☒ **Data un'espressione regolare  $E$  esiste un automa a stati finiti  $A$  che accetta il linguaggio  $L(E)$**

☒ **Dato un automa a stati finiti  $A$  che accetta il linguaggio  $L$ , esiste un'espressione regolare  $E$  tale che  $L=L(E)$**

⌘ Quindi

☒ Automi a stati finiti e espressioni regolari hanno lo stesso potere espressivo

☒ Gli automi a stati finiti possono essere usati per riconoscere se una stringa appartiene o meno ad un linguaggio regolare

# Da espressioni regolari e automi a stati finiti

⌘ La costruzione dell'automa richiede tre passi

☒ Dall'espressione regolare si costruisce un **automa non deterministico** con  $\varepsilon$ -transizioni

☒ Si trasforma l'automa non deterministico in un **automa deterministico**

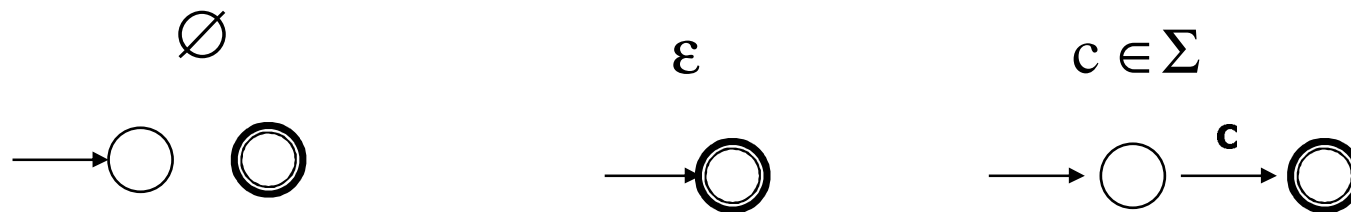
☒ Eventualmente, si trasforma ancora l'automa in modo da **minimizzare il numero degli stati**

⌘ Di seguito vedremo come funzionano tali passi

# Costruzione di automi non deterministici

⌘ La strategia di costruzione dell'automato non deterministico può essere **definita per induzione**

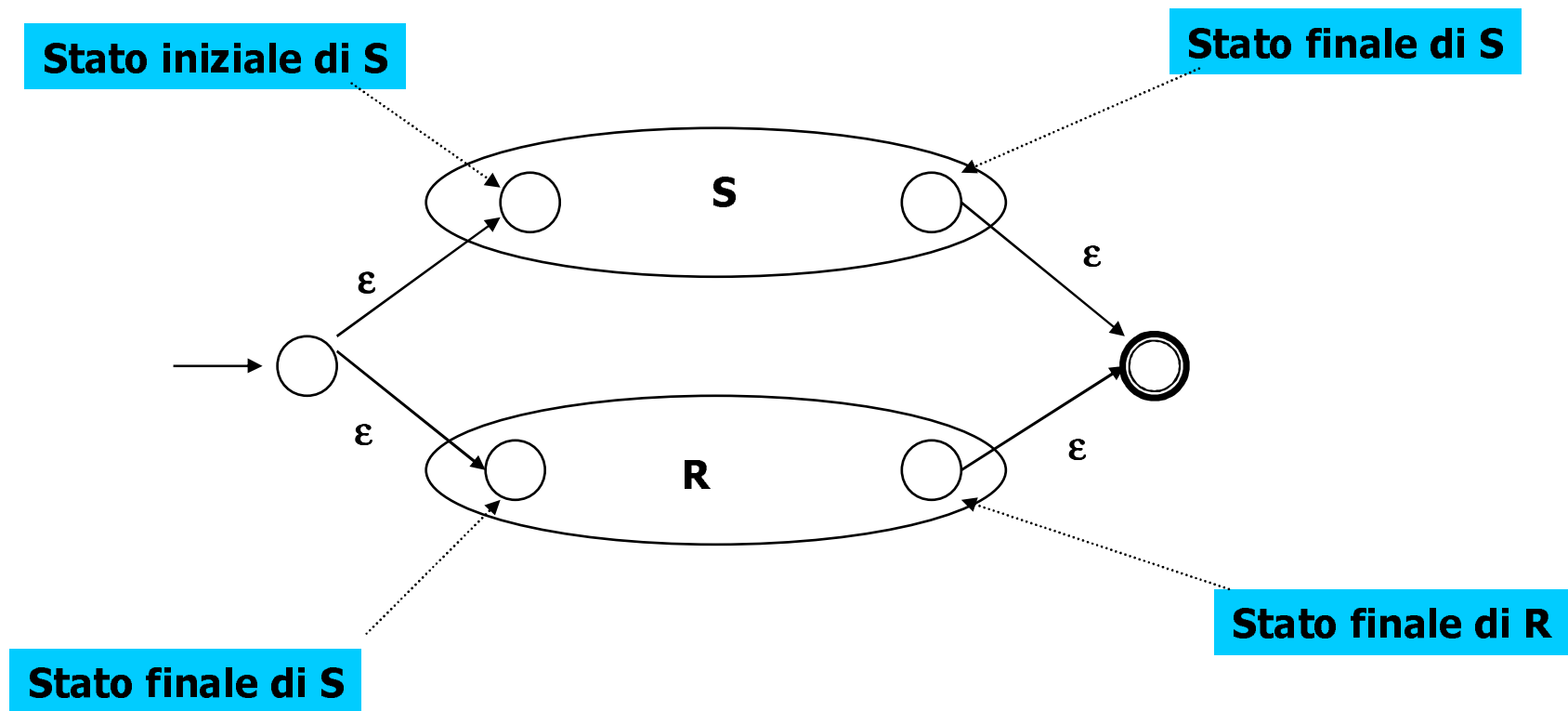
⌘ Se l'espressione  $E$  è un operando atomico  $\emptyset, \varepsilon, c \in \Sigma$



⌘ Nel seguito si suppone di conoscere come costruire l'automato per le espressioni  $R, S$  e si mostra come costruire quelle per  $R|S, RS$  e  $R^*$

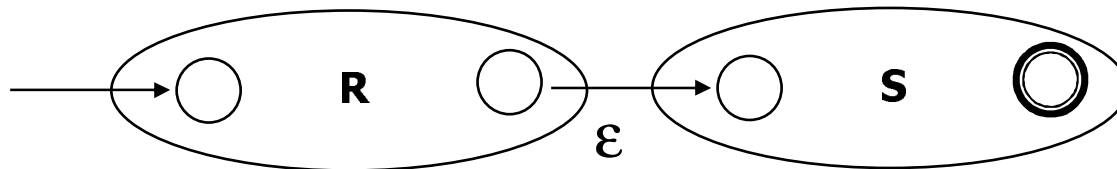
# Unione

⌘ L'espressione  $R|S$



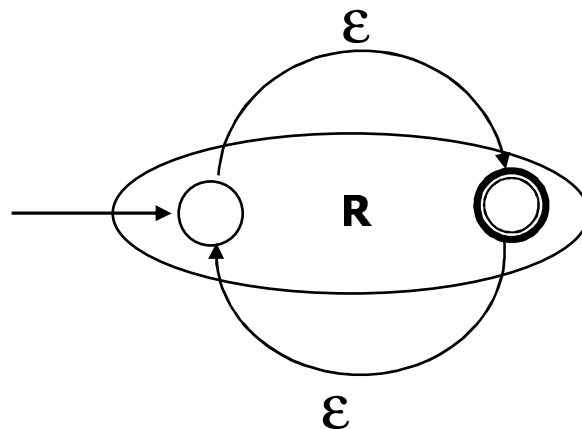
# Concatenazione e chiusura

## ⌘ L'espressione RS



- Lo stato iniziale di RS coincide con quello di R
- Lo stato finale di RS coincide con quello di S

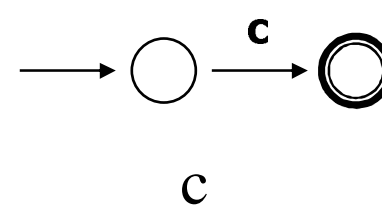
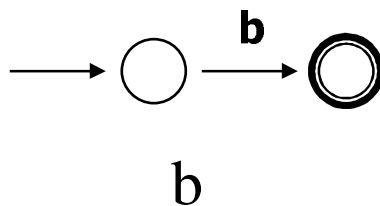
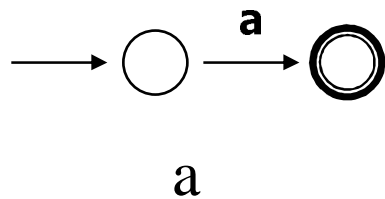
## ⌘ L'espressione $R^*$



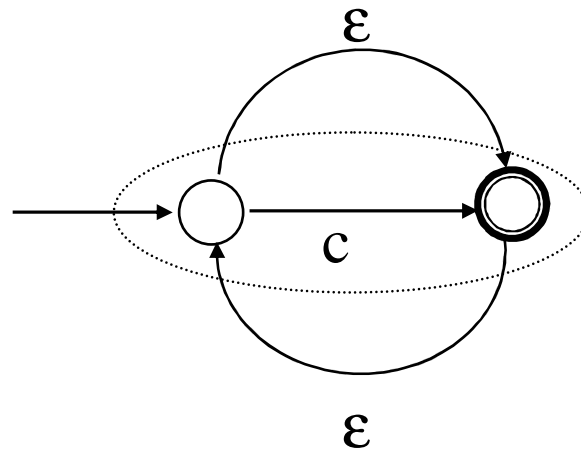
Gli stati iniziali e finali di  $R^*$  coincidono con quelli di R

# Un esempio: $a|bc^*$

⌘ Le espressioni atomiche

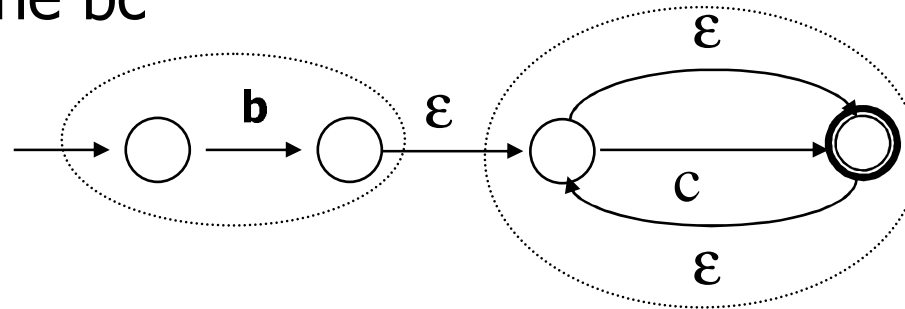


⌘ L'espressione  $c^*$

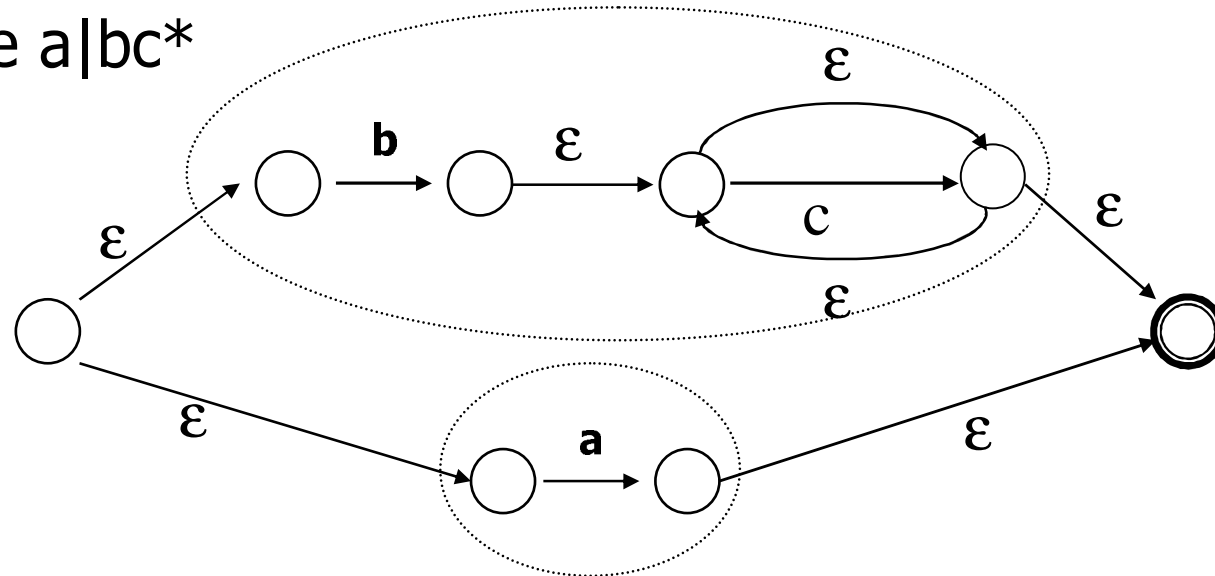


# Un esempio: $a|bc^*$

⌘ L'espressione  $bc^*$



⌘ L'espressione  $a|bc^*$





# Da automa non deterministico ad automa deterministico

⌘ E' possibile trasformare un automa non deterministico (NFA) in uno deterministico (DFA)

☒ Gli stati del DFA corrispondono a **insiemi di stati** del NFA

⌘ Si definisce l' $\epsilon$ -chiusura di un insieme di stati  $\{s_1, s_2, \dots, s_n\}$  del NFA

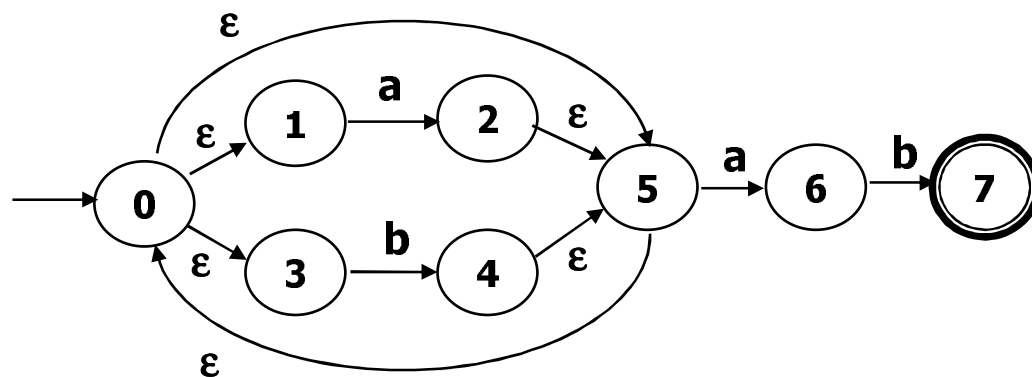
☒  $\epsilon\text{-chiusura}(s) = \{\text{gli stati raggiungibili da } s \text{ con } \epsilon\text{-transizioni}\}$

☒  $\epsilon\text{-chiusura}(\{s_1, s_2, \dots, s_n\}) = \bigcup_n \epsilon\text{-chiusura}(s_n)$

⌘ Si definisce la funzione transizione

☒  $\text{tr}(\{s_1, s_2, \dots, s_n\}, a) = \{\text{gli stati raggiungibili con una transizione } a \text{ da uno degli stati } s_1, s_2, \dots, s_n\}$

# L' $\epsilon$ -chiusura



$(a|b)^*ab$

## ⌘ Esempi

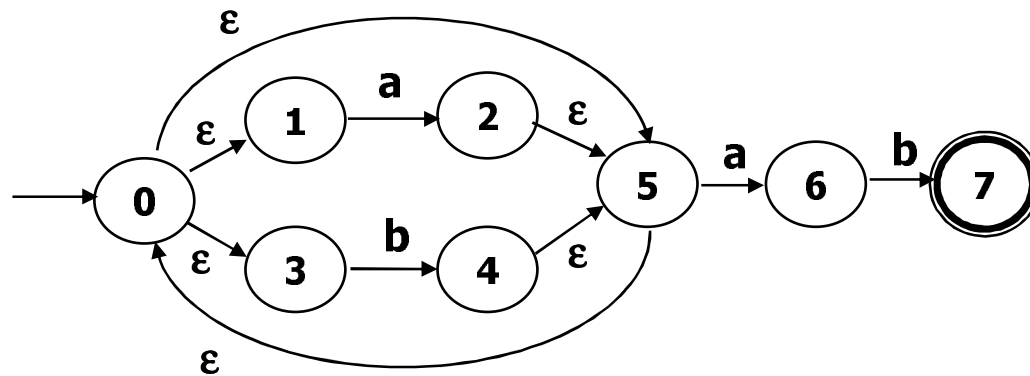
☒  $\epsilon$ -chiusura(0)={0,1,3,5},  $\epsilon$ -chiusura(1)={1},  $\epsilon$ -chiusura(2)={2,5,0,1,3}

☒  $\epsilon$ -chiusura(3)={3},  $\epsilon$ -chiusura(4)={4,5,0,1,3},  $\epsilon$ -chiusura(5)={5,0,1,3}

☒  $\epsilon$ -chiusura(6)={6},  $\epsilon$ -chiusura(7)={7}

☒  $\epsilon$ -chiusura({5,4})= $\epsilon$ -chiusura(4)  $\cup$   $\epsilon$ -chiusura(5)={4,5,0,1,3} $\cup$ {5,0,1,3}  
={4,5,0,1,3}

# La funzione tr



$(a|b)^*ab$

## ⌘ Esempi

⌘  $\text{tr}(0,a) = \emptyset, \text{tr}(1,a) = \{2\}, \text{tr}(2,a) = \emptyset, \text{tr}(3,a) = \emptyset, \dots$

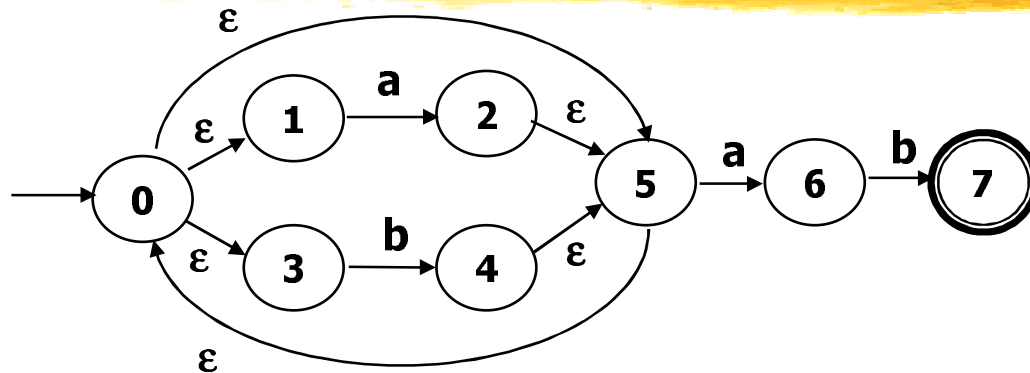
⌘  $\text{tr}(0,b) = \emptyset, \text{tr}(1,b) = \emptyset, \text{tr}(2,b) = \emptyset, \text{tr}(3,b) = \{4\}, \dots$

⌘

# Da automa non deterministico ad automa deterministico II

```
⌘ Inizializza il DFA con lo stato  $\epsilon$ -chiusura(0)
⌘ For each S=(stato non ancora marcato del DFA) do
  ⌘ marca S
  ⌘ For each a=simbolo do
    ⌘ R=  $\epsilon$ -chiusura(tr(S,a))
    ⌘ Elimina stati non importanti da R
    ⌘ if (R non esiste) then aggiungi lo stato R al DFA
    ⌘ aggiungi una transizione a dallo stato S allo stato R
    ⌘ end
  ⌘ end
```

# Un esempio



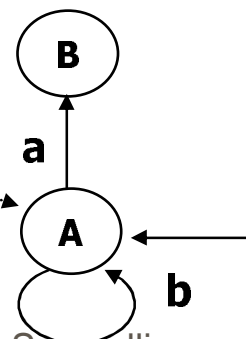
Si considerano solo gli stati importanti (si escludono gli stati che hanno solo  $\epsilon$ -transizioni uscenti)

⌘  $A = \epsilon\text{-chiusura}(0) = \{0, 1, 3, 5\} \equiv \{1, 3, 5\}$

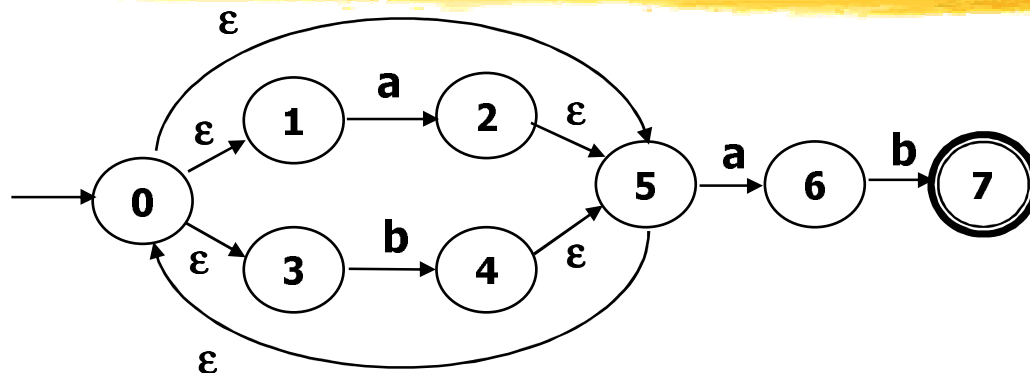
⌘  $B = \epsilon\text{-chiusura}(\text{tr}(A, a)) = \epsilon\text{-chiusura}(\{2, 6\}) = \{2, 5, 0, 1, 3, 6\} \equiv \{5, 1, 3, 6\}$

⌘  $\epsilon\text{-chiusura}(\text{tr}(A, b)) = \epsilon\text{-chiusura}(\{4\}) = \{4, 5, 0, 1, 3\} \equiv \{5, 1, 3\} = A$

Lo stato iniziale è  $\epsilon\text{-chiusura}(0)$



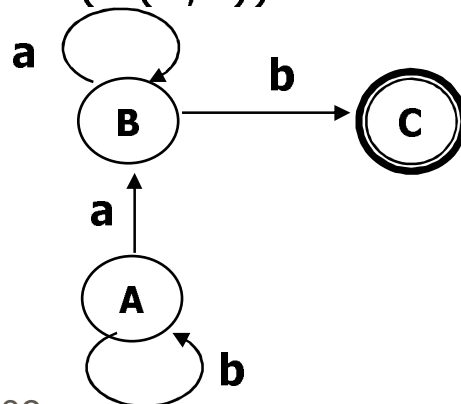
# Un esempio II



⌘  $A = \{1, 3, 5\}, B = \{5, 1, 3, 6\}$

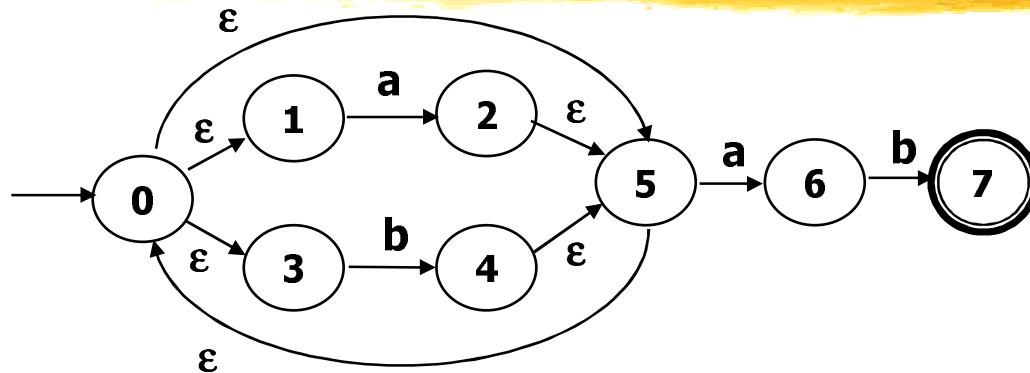
⌘  $\epsilon\text{-chiusura}(\text{tr}(B, a)) = \epsilon\text{-chiusura}(\{6, 2\}) = \{6, 2, 5, 0, 1, 3\} \equiv B$

⌘  $C = \epsilon\text{-chiusura}(\text{tr}(B, b)) = \epsilon\text{-chiusura}(\{4, 7\}) = \{4, 7, 5, 0, 1, 3\} \equiv \{7, 5, 1, 3\}$



Ogni stato contenente uno stato finale del NFA è finale nel DFA

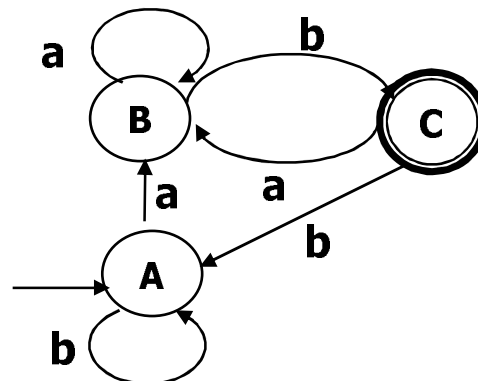
# Un esempio III



⌘  $A = \{1, 3, 5\}$ ,  $B = \{5, 1, 3, 6\}$ ,  $C = \{7, 5, 1, 3\}$

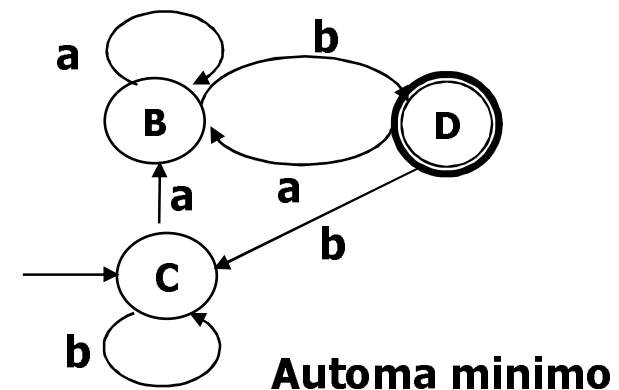
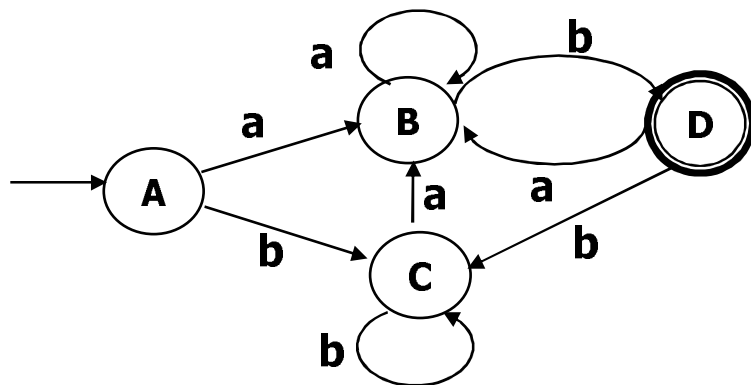
⌘  $\epsilon\text{-chiusura}(\text{tr}(C, a)) = \epsilon\text{-chiusura}(\{6, 2\}) = \{6, 2, 5, 0, 1, 3\} \equiv \{6, 5, 1, 3\} = B$

⌘  $\epsilon\text{-chiusura}(\text{tr}(C, b)) = \epsilon\text{-chiusura}(\{4\}) = \{4, 5, 0, 1, 3\} \equiv \{5, 1, 3\} = A$



# Trovare l'automa minimo

- ⌘ L'automa prodotto potrebbe non essere quello minimo (che ha il numero minimo di stati)
- ⌘ Si può dimostrare che per ogni linguaggio **l'automa minimo è unico**
- ⌘ alcuni stati possono essere equivalenti, nell'esempio  $A \equiv C$

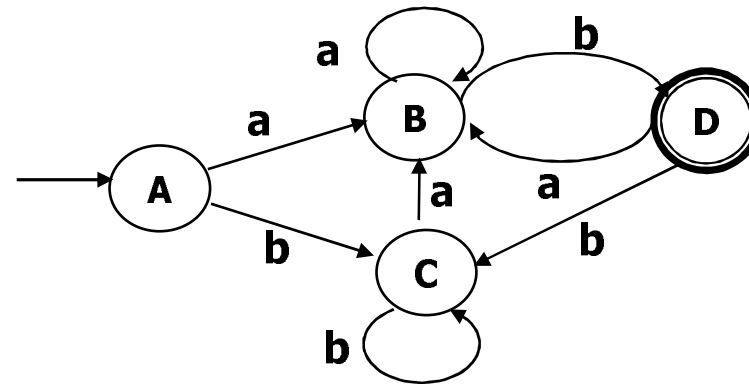




# Trovare gli stati equivalenti

- ⌘ L'algoritmo inizia con una partizione  $P=\{F,A\}$  degli stati che contiene l'insieme degli stati finali (F) e quello degli altri stati (A)
- ⌘ **While** Esiste S e simboli a e stati  $s1, s2 \in S$  tali che
$$\text{tr}(s1,a) \in U1, \text{tr}(s2,a) \in U2, U1 \neq U2$$
  - 📐 suddividi S in insiemi  $A1, A2, \dots, An$  tali che  $\text{tr}(Ai,c)=Ui$ , per ogni c

# Un esempio



⌘ All'inizio  $P = \{\{A, B, C\}, \{D\}\}$

⌘ Si considera  $\{A, B, C\}$  e il simbolo  $b$

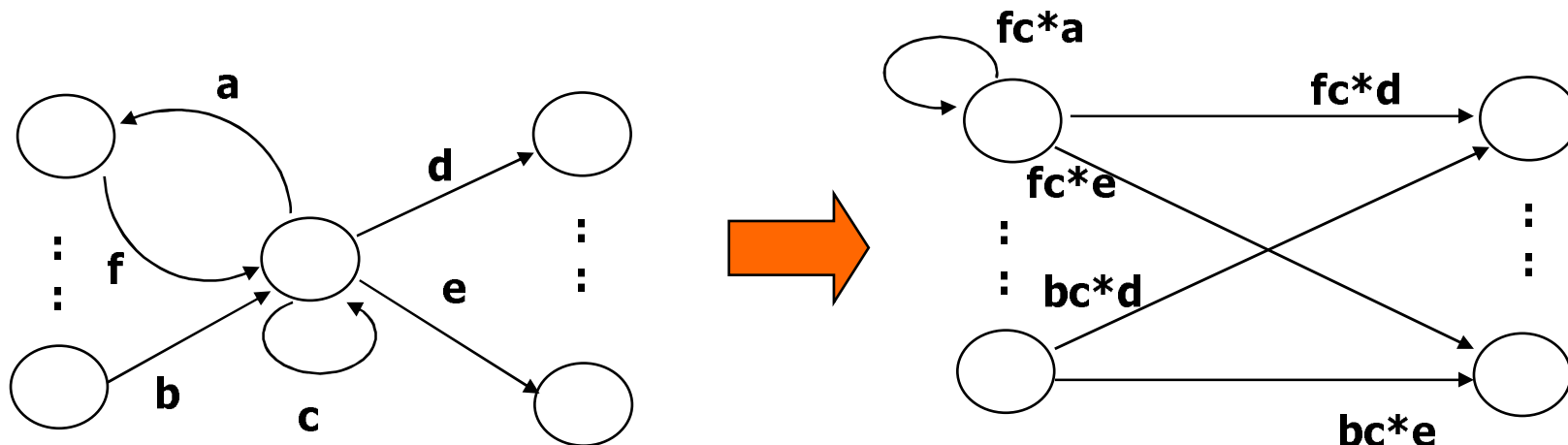
⌘ Si osserva che  $\text{tr}(B, b) = \{D\}$ , mentre  $\text{tr}(\{A, C\}, b) = \{C\}$

⌘ Quindi  $P = \{\{A, C\}, \{B\}, \{D\}\}$

⌘ A questo punto l'algoritmo si può fermare!!

# Da automa a espressione regolare

- ⌘ Dato un automa è possibile generare l'espressione regolare che esso accetta
- ⌘ Occorre rimuovere, ad uno ad uno, tutti gli stati dell'automata

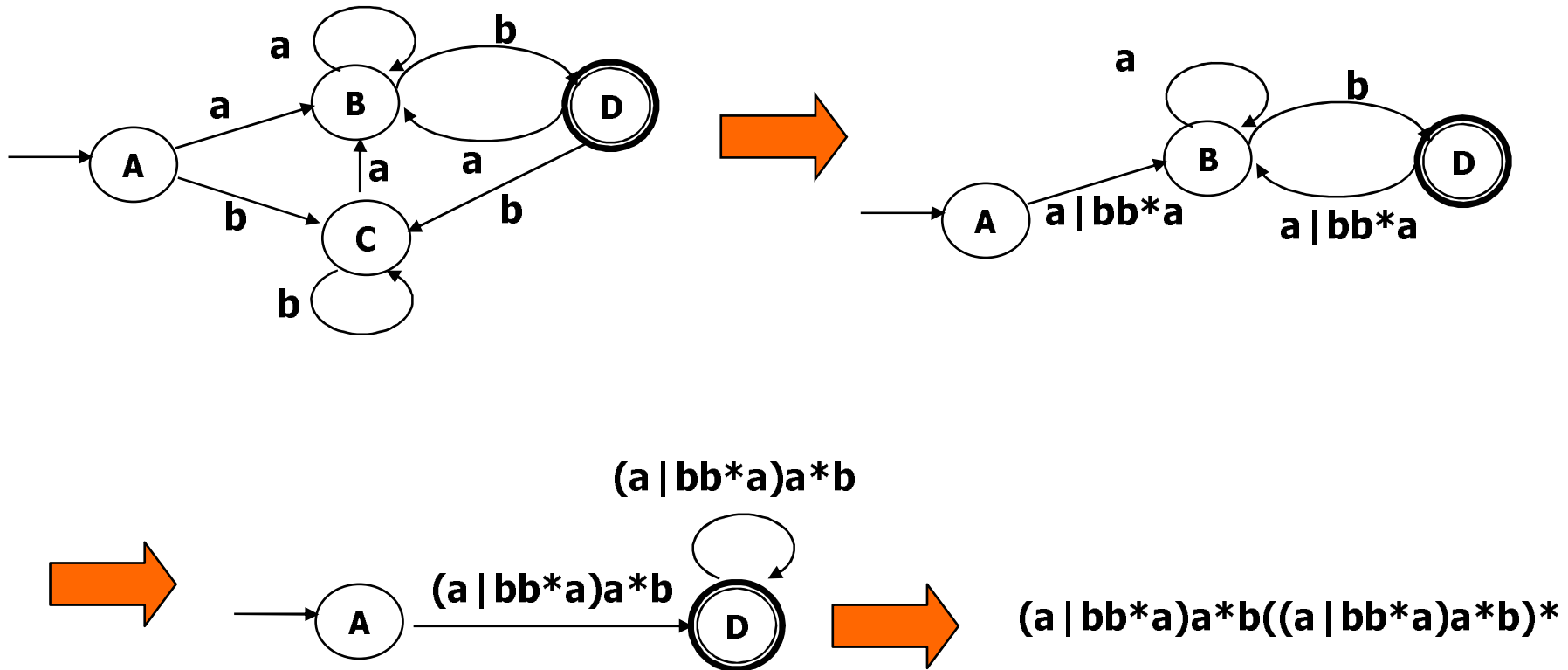


# Da automa a espressione regolare II

- ⌘ Si considerano **tutti** gli stati finali, **uno alla volta**
- ⌘ Si **rimuovono gli stati** dell'automa fino quando non rimangono solo quello iniziale e quello finale scelto



# Un esempio



# Dove si usano le espressioni regolari ?

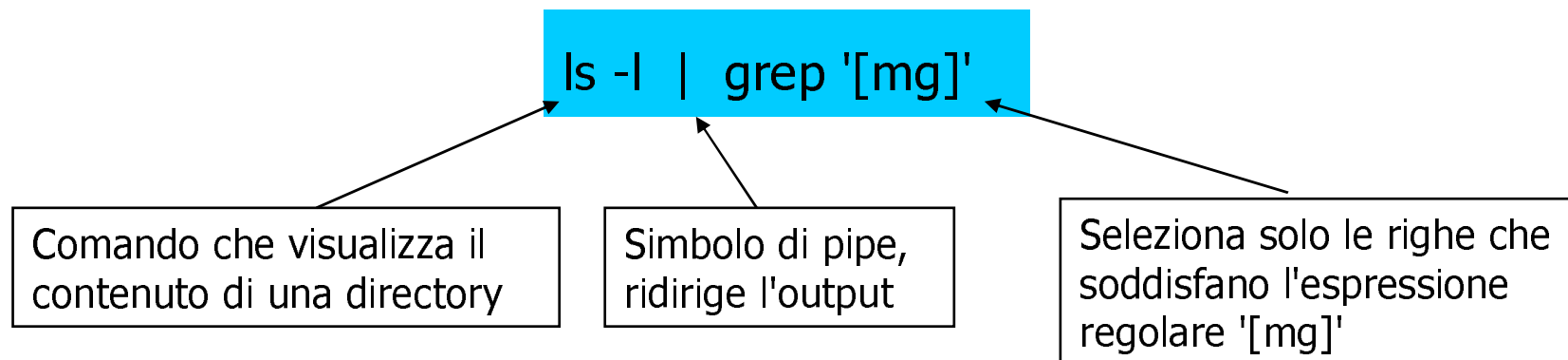


- ⌘ Le espressioni regolari sono usate per
  - ☒ generazione di **analizzatori lessicali**
  - ☒ comandi di ricerca **negli editor**
  - ☒ **funzioni di libreria** che implementano il matching con espressioni regolari
  - ☒ **comandi** per la ricerca di pattern
- ⌘ Molti di questi programmi sono stati sviluppati originariamente in Unix

# Esempio: il comando grep

⌘ Il comando Unix **grep** permette di realizzare un filtro con cui selezionare alcune righe di un testo

**Questo comando Unix mostra il contenuto di una directory selezionando solo alcune righe**



# Posix 1003.2

⌘ Posix 1003.2 definisce uno standard con cui si specificano le espressioni regolari

⌘ Operatori

☒  $E?$  (L'espressione  $E$  è opzionale)

☒  $E^*$  (Chiusura di Kleene)

☒  $E^+$  (L'espressione  $E$  è ripetuta almeno una volta).

☒  $E\{n\}$  (L'espressione  $E$  è ripetuta esattamente  $n$  volte)

☒  $^E$  (indica che  $E$  appare all'inizio di una riga)

☒  $E\$$  (indica che  $E$  appare alla fine di una riga)



# Posix 1003.2 II



## ⌘ Atomi

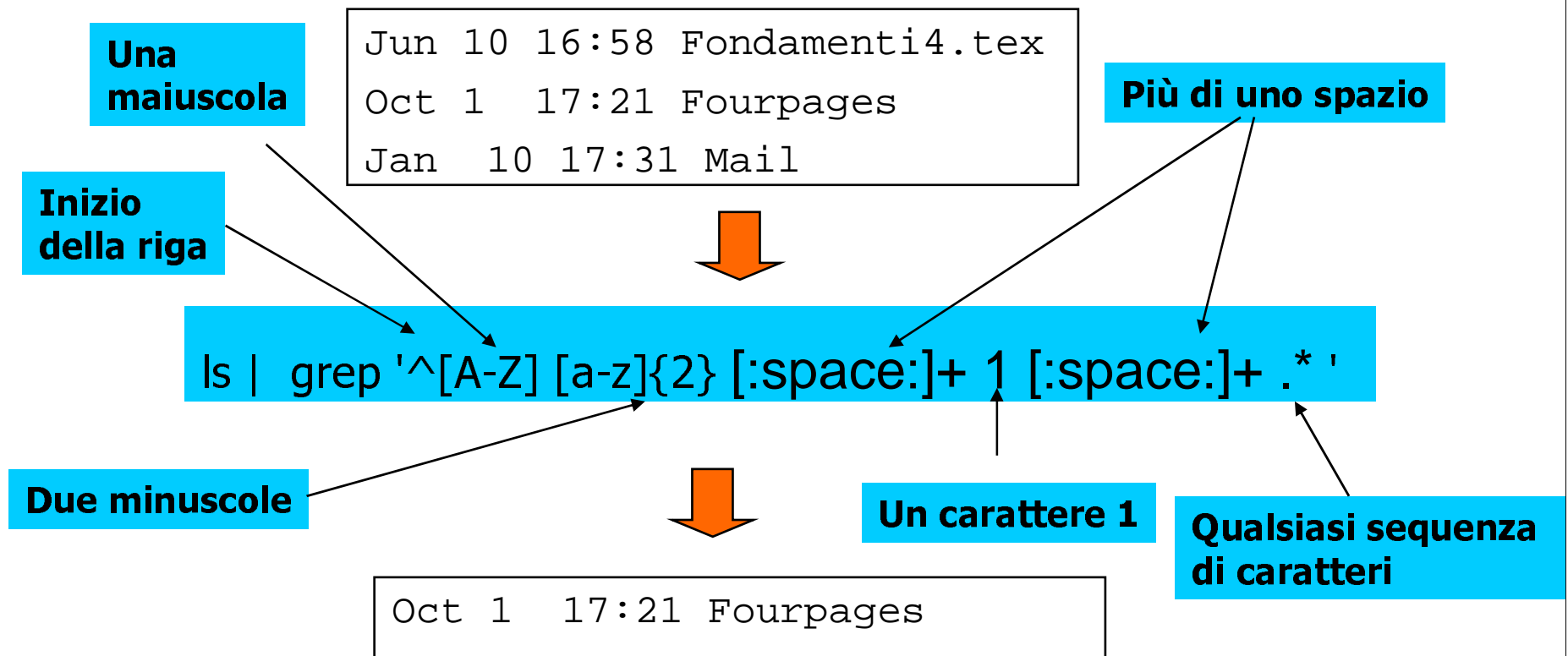
- ⌘. (il punto indica un qualsiasi carattere)
- ⌘[abcd] (qualsiasi carattere fra a, b, c, d)
- ⌘[0-9] (le cifre, il simbolo "-" specifica che si devono prendere tutti in caratteri compresi fra 0 e 9)
- ⌘[a-zA-Z] (tutte le lettere, maiuscole e minuscole)

## ⌘ Classi predefinite

- ⌘[:alpha:] (qualsiasi lettera)
- ⌘[:digit:] (qualsiasi cifra)
- ⌘[:space:] (qualsiasi carattere di spaziatura)

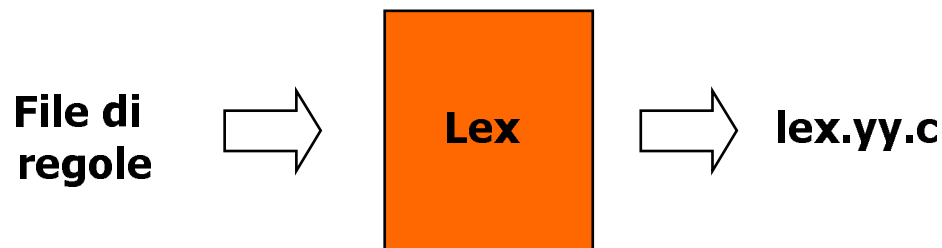
# Un esempio

⌘ Questo comando seleziona tutti i file modificati il primo del mese



# Generazione di analizzatori lessicali (Lex)

- ⌘ In pratica, esistono strumenti che facilitano la realizzazione di analizzatori lessicali
- ⌘ Lex è un generatore di analizzatori lessicali
- ⌘ Lex prende in ingresso un file che contiene un insieme di regole e genera un file C che implementa un analizzatore lessicale



# File di regole Lex

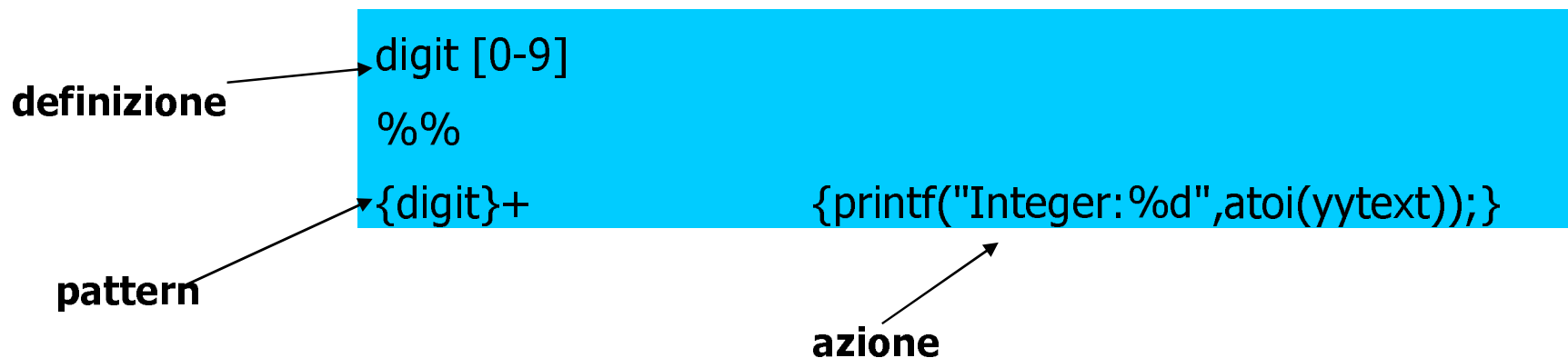
⌘ Un file di regole Lex è costituito da una parte di **definizioni** e una di **regole**

⌘ Le regole sono costituite da un **pattern** e un'**azione**

⊞ il pattern è un'espressione regolare

⊞ l'azione è codice C

⊞ l'analizzatore esegue il comando quando riconosce il pattern



# Un esempio: riconoscere identificatori e parole chiave

<b>Definizioni</b>	digit [0-9] id [a-z][a-z0-9]*	
<b>inizio regole</b>	%%	
<b>interi</b>	{digit}+	{printf("Integer: %d",atoi(yytext));}
<b>reali</b>	{digit}+"."{digit}+	{printf("Float: %g",atof(yytext));}
<b>parole chiave</b>	if then begin end	{printf("Keyword: %s",yytext);}
<b>identificatori</b>	{id}	{printf("Identifier: %s",yytext);}
<b>operatori</b>	"+" "-" "/" "*"	{printf("Operator: %s",yytext);}
<b>rimuove gli spazi</b>	[ \t\n]	
<b>caratteri sconosciuti</b>	.	{printf("Unknown char.: %s",yytext);}

# **Analisi Sintattica**

# Le grammatiche libere da contesto

- ⌘ Le **grammatiche libere** da contesto (context free) sono uno strumento con cui si possono definire i principali linguaggi di programmazione
- ⌘ Una grammatica libera da contesto è definita da
  - ☒ Un insieme di **simboli terminali** che definiscono i simboli atomici del linguaggio
  - ☒ Un insieme di **simboli non terminali** (variabili) che definiscono le categorie sintattiche
  - ☒ Un insieme di **produzioni** che sono regole che definiscono come si possono generare le stringhe del linguaggio

# Le grammatiche libere da contesto: un esempio

⌘ La seguente grammatica genera le espressioni aritmetiche

## regole

$E \rightarrow n$   
 $E \rightarrow (E)$   
 $E \rightarrow E + E$   
 $E \rightarrow E - E$   
 $E \rightarrow E * E$   
 $E \rightarrow E / E$

## Simboli terminali

$T = \{ (, ), +, -, *, /, n \}$

## Simboli non terminali

$NT = \{ E \}$

Il simbolo  $n$  indica un numero

## Esempi di espressioni generate dalla grammatica

$n * n - n$   
 $(n + n) * n$   
 $(n - n) * (n + n)$

## La derivazione di un'espressione aritmetica

$E \rightarrow E - E \rightarrow E * E - E \rightarrow n * E - E \rightarrow n * n - E \rightarrow n * n - n$



# Le grammatiche libere da contesto II

⌘ Formalmente una grammatica libera da contesto è una quadrupla  $G=(V,T,P,S)$

☒  $V$  è un insieme di **simboli non terminali**

☒  $T$  un insieme di **simboli terminali**

☒  $P$  un insieme di **produzioni** del tipo

$A \rightarrow \alpha$ , dove  $A \in V$  e  $\alpha \in (V \cup T)^*$

☒  $S \in V$  è lo scopo o **simbolo di partenza**

# Le grammatiche libere da contesto III

## ⌘ Una **derivazione diretta**

$$\boxed{\wedge} \beta A \gamma \Rightarrow \beta \alpha \gamma$$

$$\boxed{\wedge} \text{dove } \beta, \alpha, \gamma \in (V \cup T)^* \text{ e } A \rightarrow \alpha \in P$$

## ⌘ Una **derivazione**

$$\boxed{\wedge} \alpha_1 \Rightarrow^* \alpha_n$$

$$\boxed{\wedge} \text{dove } \alpha_1, \dots, \alpha_n \in (V \cup T)^* \text{ e } \alpha_1 \Rightarrow \alpha_2 \dots \alpha_{n-1} \Rightarrow \alpha_n$$

## ⌘ Il **linguaggio $L(G)$ generato** da una grammatica

$$\boxed{\wedge} L(G) = \{ \alpha \mid \alpha \in T^*, S \Rightarrow^* \alpha \}$$

# Un esempio

⌘ In generale determinare cosa genera una grammatica è **un problema difficile**

Una grammatica che genera tutte le stringhe con lo stesso numero di a e b

$V = \{S, A, B\}$

$T = \{a, b\}$

$S \rightarrow aB \quad A \rightarrow bA$

$S \rightarrow bA \quad B \rightarrow b$

$A \rightarrow a \quad B \rightarrow bS$

$A \rightarrow aS \quad B \rightarrow aBB$

Ipotesi

⏏  $S \rightarrow \alpha$  se e solo se  $\alpha$  ha lo stesso numero di a e b

⏏  $A \rightarrow \alpha$  se e solo se  $\alpha$  ha un numero di a maggiore di uno rispetto al numero dei b

⏏  $A \rightarrow \alpha$  se e solo se  $\alpha$  ha un numero di b maggiore di uno rispetto al numero degli a

Dimostrazione per induzione su  $|\alpha|$

⏏ se  $|\alpha| = 1$ , le uniche produzioni possibili sono  $A \rightarrow a$  e  $B \rightarrow b$

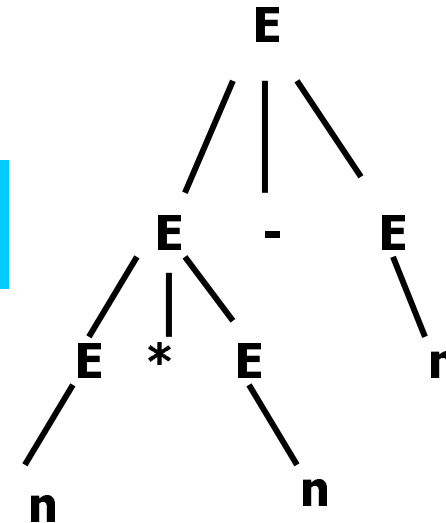
⏏ se  $|\alpha| > 1$ , si considera la prima produzione di  $A \Rightarrow^* \alpha$

⏏ se è  $S \rightarrow aB$ , allora  $aB$  contiene un a in più .....

# Alberi sintattici e derivazioni

- ⌘ Un **albero sintattico** (o albero di analisi) è una rappresentazione di una derivazione
- ⌘ Ogni nodo corrisponde ad una produzione
- ⌘ L'albero rappresenta quali produzioni sono state usate, ma **non l'ordine** in cui sono state usate

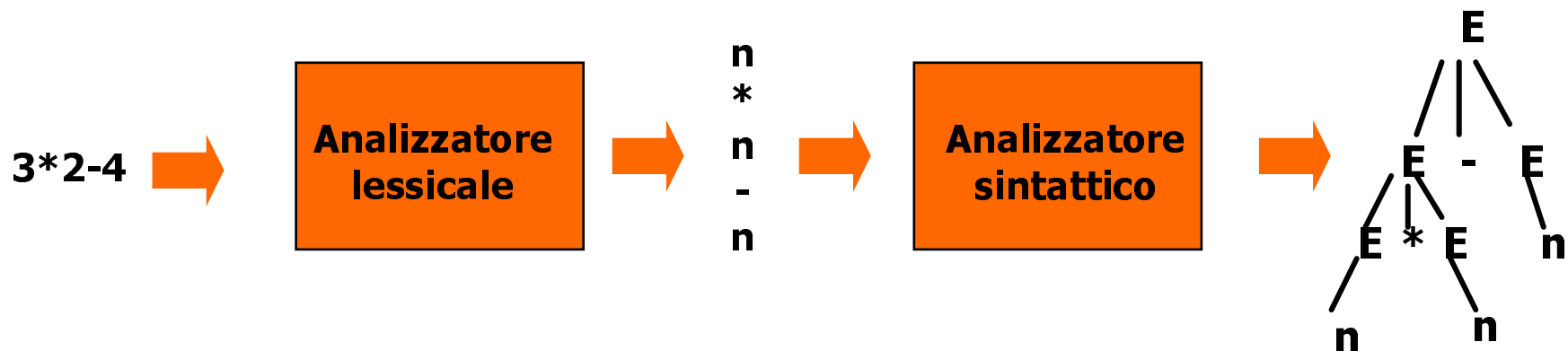
$E \rightarrow E-E \rightarrow E * E-E \rightarrow n * E-E \rightarrow n * n-E \rightarrow n * n-n$



# Come cooperano l'analizzatore lessicale e quello sintattico

⌘ Nei comuni compilatori

- ☒ l'analizzatore lessicale riconoscere i **token** (simboli terminali) del linguaggio: numeri interi, reali, variabili, parole chiave, operatori, ...
- ☒ l'analizzatore lessicale invia all'analizzatore sintattico la sequenza dei simboli terminali individuati
- ☒ l'analizzatore sintattico analizza la sequenza di simboli terminali e produce l'albero sintattico



# Costruire un analizzatore sintattico dalla grammatica

⌘ Come fa un analizzatore a decidere se una stringa  $w$  appartiene al linguaggio generato dalla grammatica  $G$  ?

⌘ Analizzatori discendenti (top down)

☒ Partendo da simbolo  $S$  si tenta di ricostruire l'albero di derivazione con cui si ottiene  $w$

☒ Si usa le regole della grammatica in avanti

$S \rightarrow aABe \rightarrow aAbcBe \rightarrow abbcBe \rightarrow abbcde$

Prima il simbolo  
più a sinistra

⌘ Analizzatori ascendenti (bottom up)

☒ Partendo da  $w$  si tenta di ricondurci a  $S$

☒ Si usa le regole della grammatica all'indietro

$abbcde \rightarrow aAbcde \rightarrow aAde \rightarrow aABe \rightarrow S$

(1)  $S \rightarrow aABe$

(2)  $A \rightarrow Abc|b$

(3)  $B \rightarrow d$

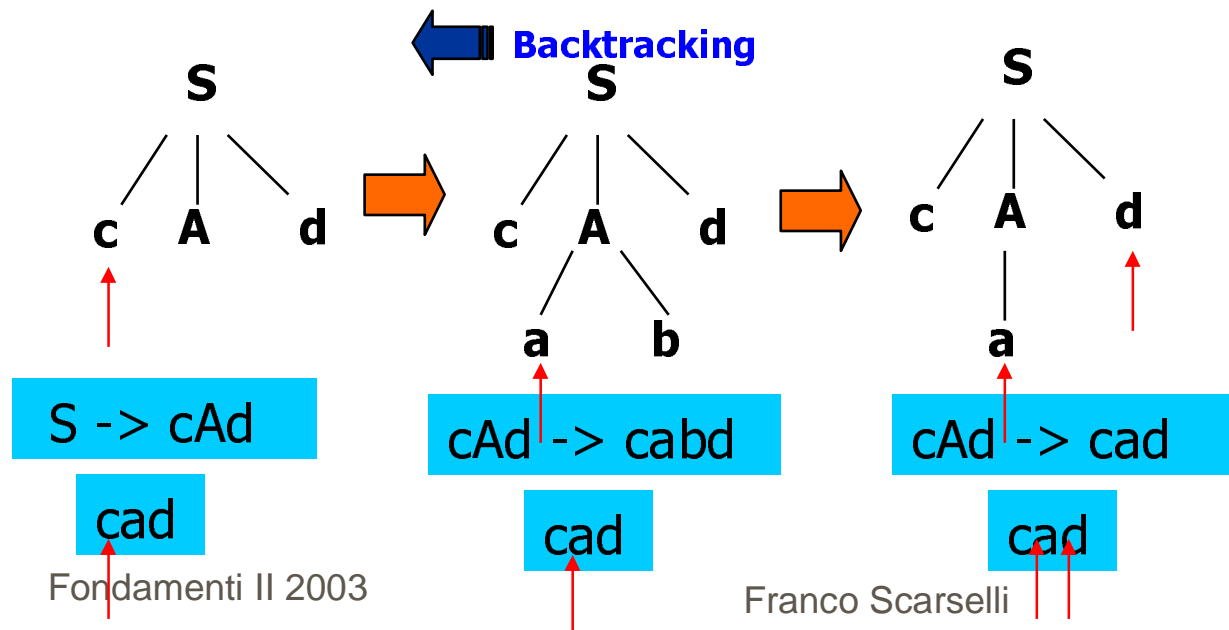
$w=abbcde$

Prima il simbolo  
più a destra

# **Analisi discendente**

# Analisi discendente

- ⌘ Si applicano le regole in maniera ricorsiva verificando se la stringa generata è compatibile con quella da riconoscere
- ⌘ Può essere necessario fare backtracking
- ⌘ La computazione finisce quando la stringa è stata generata (stringa generata) o non è possibile provare altre regole (stringa rifiutata)



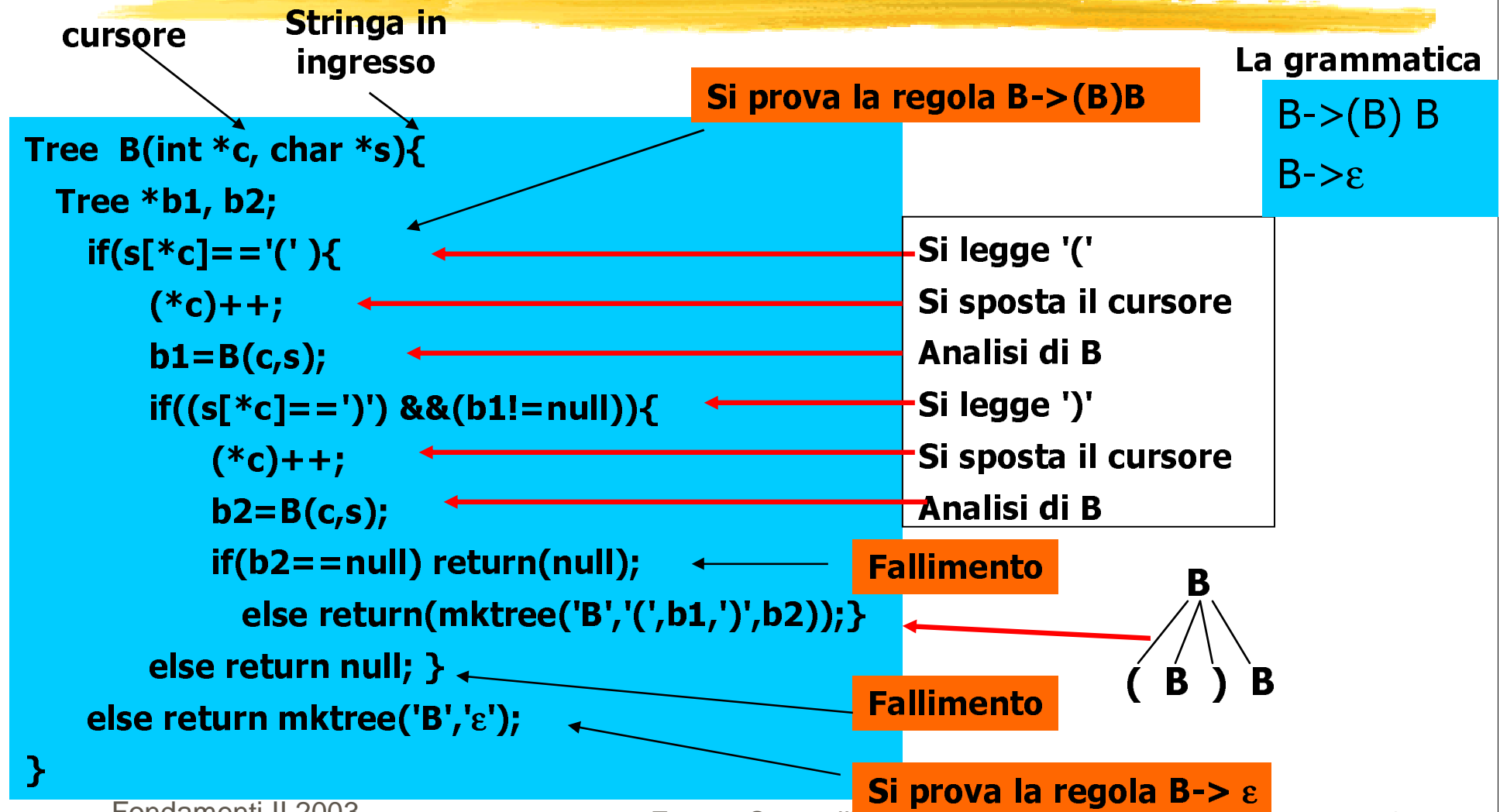
(1)  $S \rightarrow cAd$   
(2)  $A \rightarrow ab \mid a$   
 $w=cad$



# Analisi ricorsiva discendente

- ⌘ Si tiene **traccia del prossimo simbolo** da sinistra della stringa in ingresso che deve essere generato dall'albero di analisi
  - ☒ serve a limitare l'insieme di produzioni che possono essere usate
- ⌘ Si espandono i nodi **da sinistra a destra**
  - ☒ Un simbolo terminale soddisfa l'obiettivo se coincide con il prossimo simbolo in ingresso
  - ☒ Un simbolo non terminale soddisfa l'obiettivo se lo soddisfa un suo albero di analisi
- ⌘ Quando espandendo una produzione e si genera un simbolo terminale **si verifica se corrisponde con quello in ingresso**
  - ☒ Sì - Si sposta il cursore in avanti e si prosegue
  - ☒ No - Si fallisce e si prova a soddisfare l'obiettivo con un altro albero

# Analisi risorsiva discendente: un esempio



# Analizzatori predittivi

⌘ Un **analizzatore predittivo** è un analizzatore che guardando il prossimo simbolo della stringa in ingresso è in grado di

- ☒ prevedere quale regola deve essere usata ad ogni passo
- ☒ evitare il backtracking

Il simbolo terminale in ingresso:  $a$

Il simbolo non terminale:  $A$

Un insieme di regole per  $A$ :  $A \rightarrow w_1 | w_2 | \dots | w_n$



Esiste una sola regola i t.c.

$A \rightarrow w_i \rightarrow a\alpha$

**Un esempio: ad ogni passo conosco quale regola scegliere**

$S \rightarrow aSc | bSc | d$   
 $w = abdcc$

$S \rightarrow aSc$   
 $abdcc$

$\rightarrow abScc$   
 $abdcc$

$\rightarrow abdcc$   
 $abdcc$

# Analizzatori predittivi II

⌘ **Solo alcune grammatiche** permettono di realizzare analizzatori predittivi

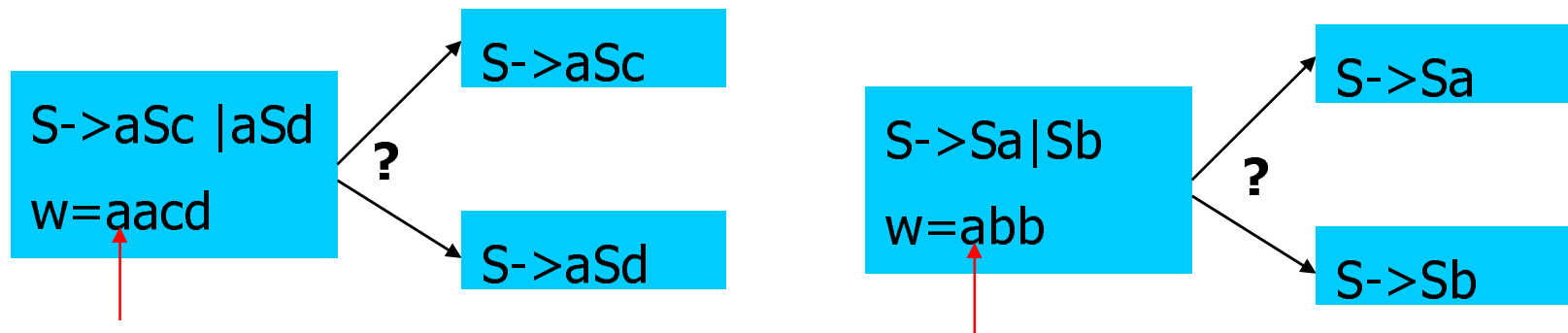
⌘ Occorre

☒ evitare grammatiche **ambigue**

☒ rimuovere la **ricorsione a sinistra**

☒ **fattorizzare la grammatica a sinistra**

Come faccio a **scegliere la regola giusta** guardando solo il primo simbolo in ingresso?



# Ambiguità

⌘ Una grammatica si dice **ambigua** se ci sono due alberi di analisi che producono la stessa stringa

Una grammatica **ambigua** che produce le parentesi bilanciate

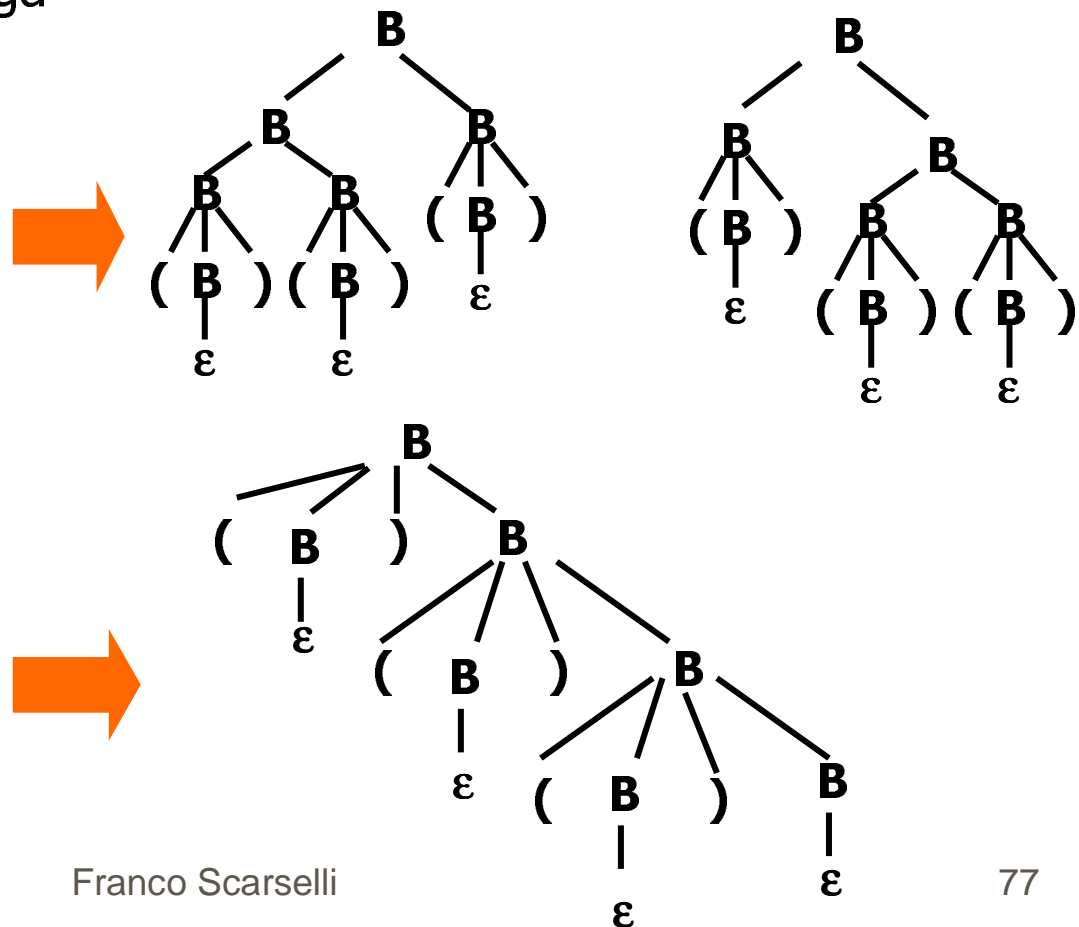
$B \rightarrow (B)$     $B \rightarrow BB$     $B \rightarrow \varepsilon$

Esempi di parentesi bilanciate

$()()()$     $((()))$     $((()))()$

Una grammatica **non ambigua**

$B \rightarrow (B) B$     $B \rightarrow \varepsilon$



# Problemi dovuti all'ambiguità

## ⌘ L'ambiguità

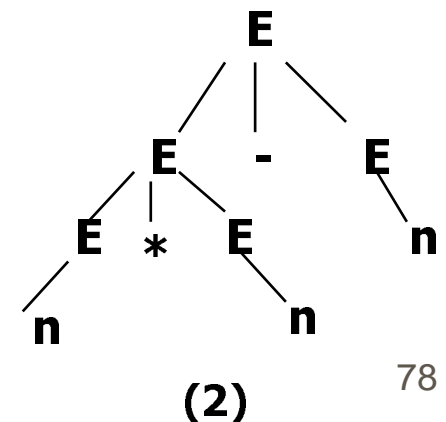
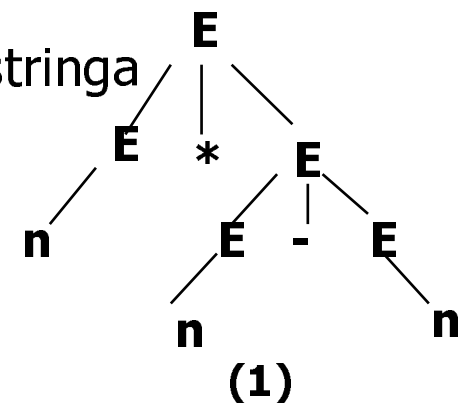
⏏ non è possibile scegliere un'unica regola di derivazione

⏏ può causare problemi nell'interpretazione di una stringa

⏏ Nell'esempio sottostante usando (1) invece di (2) **non si rispetta la precedenza** fra gli operatori

(1)  $E \rightarrow E * E \rightarrow E * E - E \rightarrow n * E - E \rightarrow n * n - E \rightarrow n * n - n$

(2)  $E \rightarrow E - E \rightarrow E * E - E \rightarrow n * E - E \rightarrow n * n - E \rightarrow n * n - n$



# Eliminazione dell'ambiguità per le espressioni aritmetiche

- ⌘ Per risolvere il problema della precedenza degli operatori occorre trovare un'altra grammatica non ambigua per lo stesso linguaggio
- ⌘ Si introducono **nuovi simboli non terminali**

⊠ E (Espressione)

⊠ T (Termine)

⊠ F (Fattore)

## Grammatica non ambigua

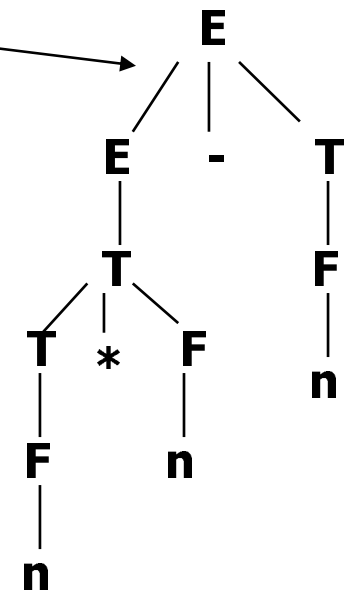
$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid n$

Un esempio  
di derivazione

$E \rightarrow E - T \rightarrow T - T \rightarrow T * F - T$   
 $\rightarrow F * F - T \rightarrow n * F - T$   
 $\rightarrow n * n - T \rightarrow n * n - F \rightarrow n * n - n$

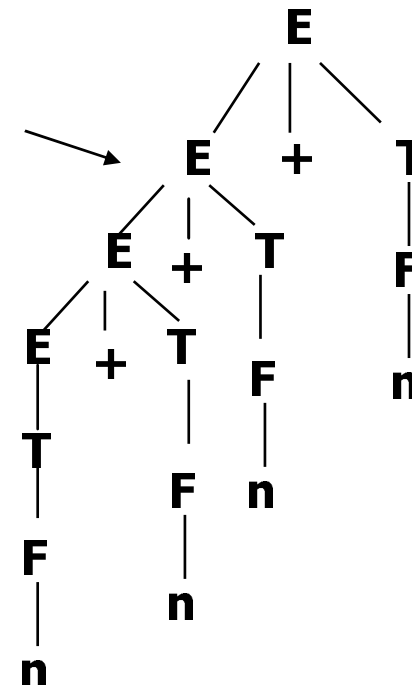


# La grammatica determina l'ordine di raggruppamento

- ⌘ La nuova grammatica usa produzioni dove la variabile da sostituire appare a sinistra nella stringa prodotta (cioè, produzioni del tipo  $A \rightarrow Aw$ )
  - 📦 le espressioni saranno valutate **raggruppando a sinistra**

$E \rightarrow E+T \mid E-T \mid T$   
 $T \rightarrow T * F \mid T / F \mid F$   
 $F \rightarrow (E) \mid n$

L'albero  
dell'espressione  
 $n+n+n+n$





# Eliminazione dell'ambiguità nel caso dell' if-then-else

- ⌘ La seguente grammatica che genera un linguaggio contenente il comando if-then-else è ambigua

**S -> if e then S**

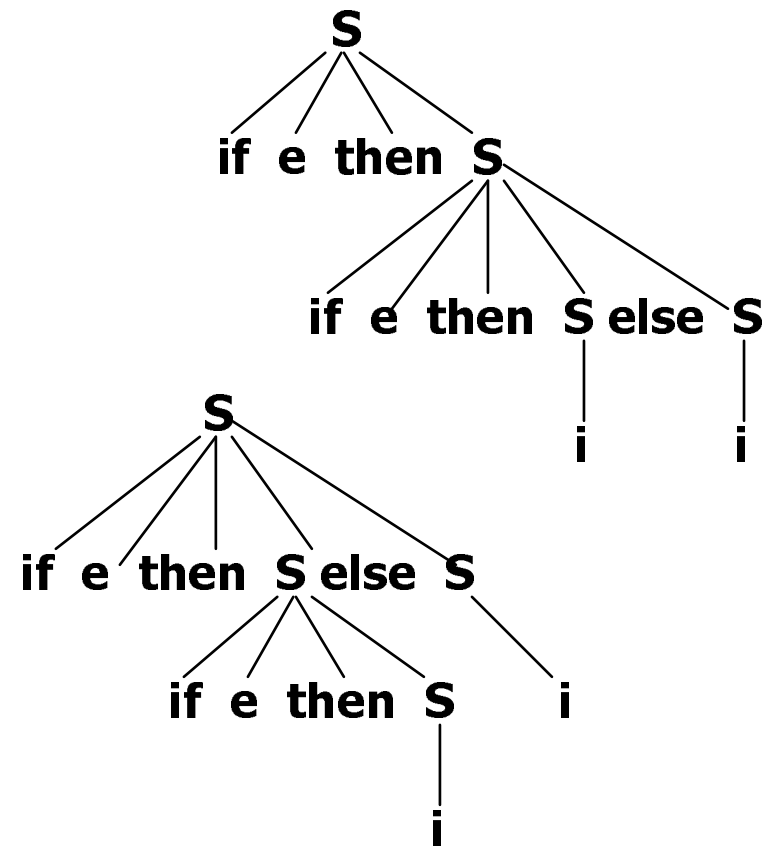
**S -> if e then S else S**

**S -> i**

Altre istruzioni

L'istruzione dopo else è legata al primo o il secondo if ?

**if e then if e then i else i**



# Eliminazione dell'ambiguità nel caso dell' if-then-else II

⌘ Per eliminare l'ambiguità si introducono nuove variabili

☒ S (uno statment)

☒ I (un statment **if** incompleto, cioè senza **else**)

☒ C (un statment **if** completo)

⌘ La grammatica associa la parte else all'if più vicino (l'ultimo)

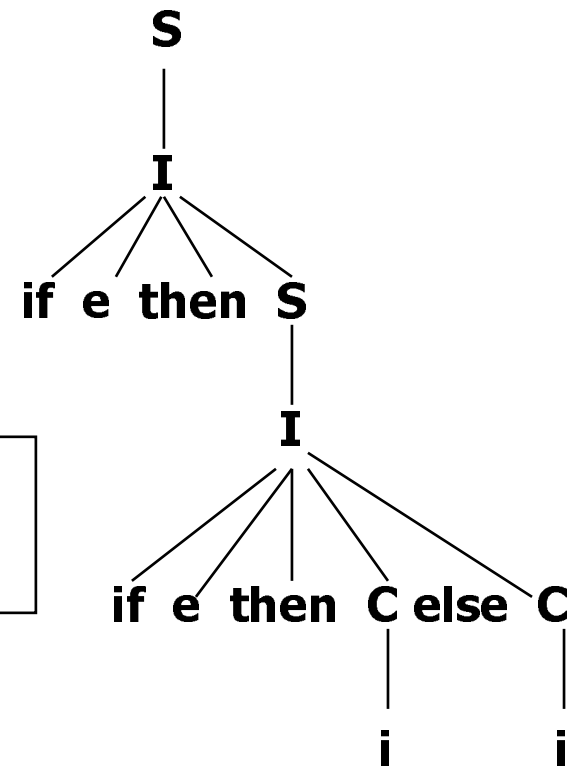
$S \rightarrow C \mid I$

$C \rightarrow \text{if } e \text{ then } C \text{ else } C \mid i$

$I \rightarrow \text{if } e \text{ then } S \mid$

$\text{if } e \text{ then } C \text{ else } I$

Fra un then e un else  
ci può essere solo uno  
statment completo



# Ricorsione a sinistra

⌘ Una grammatica si dice **ricorsiva a sinistra** se

☒ esiste una derivazione t.c.  $A \Rightarrow^* A\alpha$ , dove  $\alpha \in (V \cup T)^*$

⌘ Nel caso più semplice la grammatica contiene una produzione  $A \rightarrow A\alpha$   
questa viene detta **ricorsione immediata**

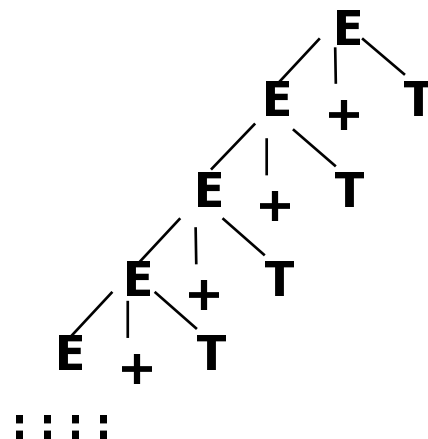
⌘ Il problema è che non si capisce **quando fermare l'espansione** del  
non terminale

$E \rightarrow E+T \mid T$

$E \rightarrow T * F \mid F$

$F \rightarrow (E) \mid n$

$w = n+n+n$



Quando  
mi fermo ?

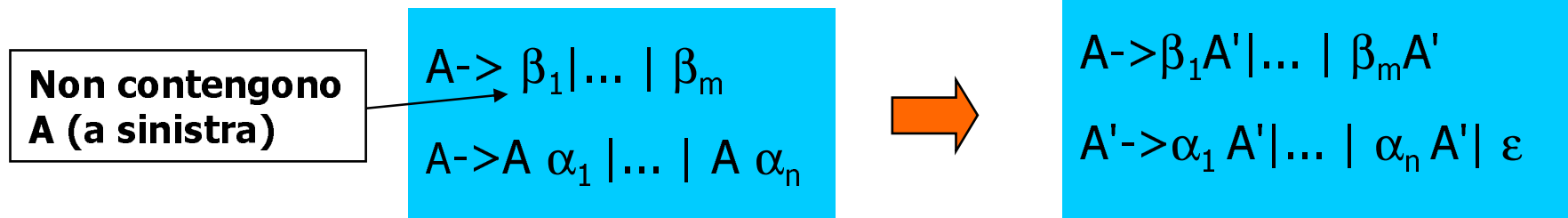
$E \rightarrow E+T \rightarrow E+E+T \rightarrow E+E+E+T \rightarrow \dots$

# Eliminare la ricorsione a sinistra

⌘ Per eliminare la ricorsione sinistra immediata

☒ Si trasforma la ricorsione da sinistra a destra

☒ Occorre introdurre un nuovo simbolo non terminale



# Eliminazione della ricorsione a sinistra: un esempio

⌘ L'eliminazione della ricorsione nel caso delle espressioni aritmetiche

$E \rightarrow E+T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid n$



$E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \varepsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid \varepsilon$   
 $F \rightarrow (E) \mid n$

⌘ Si osservi che

⏏ nel caso in cui gli  $\alpha, \beta$  della produzione ( $A \rightarrow A \alpha \mid \beta$ ) contengano copie di  $A$ , può essere necessario iterare la trasformazione

⏏ abbiamo descritto come rimuovere la ricorsione immediata, ma esiste anche un **algoritmo per la ricorsione non immediata**

# Fattorizzazione a sinistra

- ⌘ Una grammatica è **fattorizzabile a sinistra** se una variabile ha due produzioni che la espandono in stringhe con lo stesso prefisso

$$A \rightarrow \alpha \beta \mid \alpha \gamma$$

- ⌘ Il problema è: come si espande A quando vedo  $\alpha$  ?

- ⌘ Per fattorizzare a sinistra la grammatica

- ☒ Si raggruppano le regole che generano lo stesso prefisso
- ☒ Si introduce un nuovo simbolo per generare la parte destra della regola

$$A \rightarrow \alpha \beta_1 \mid \dots \mid \alpha \beta_m \mid \gamma$$



$$A \rightarrow \alpha A' \mid \gamma$$

$$A' \rightarrow \beta_1 \mid \dots \mid \beta_m$$

# Fattorizzazione a sinistra: un esempio

⌘ Nel caso dell'if-then-else si introduce un nuovo non terminale E che rappresenta la parte **else**

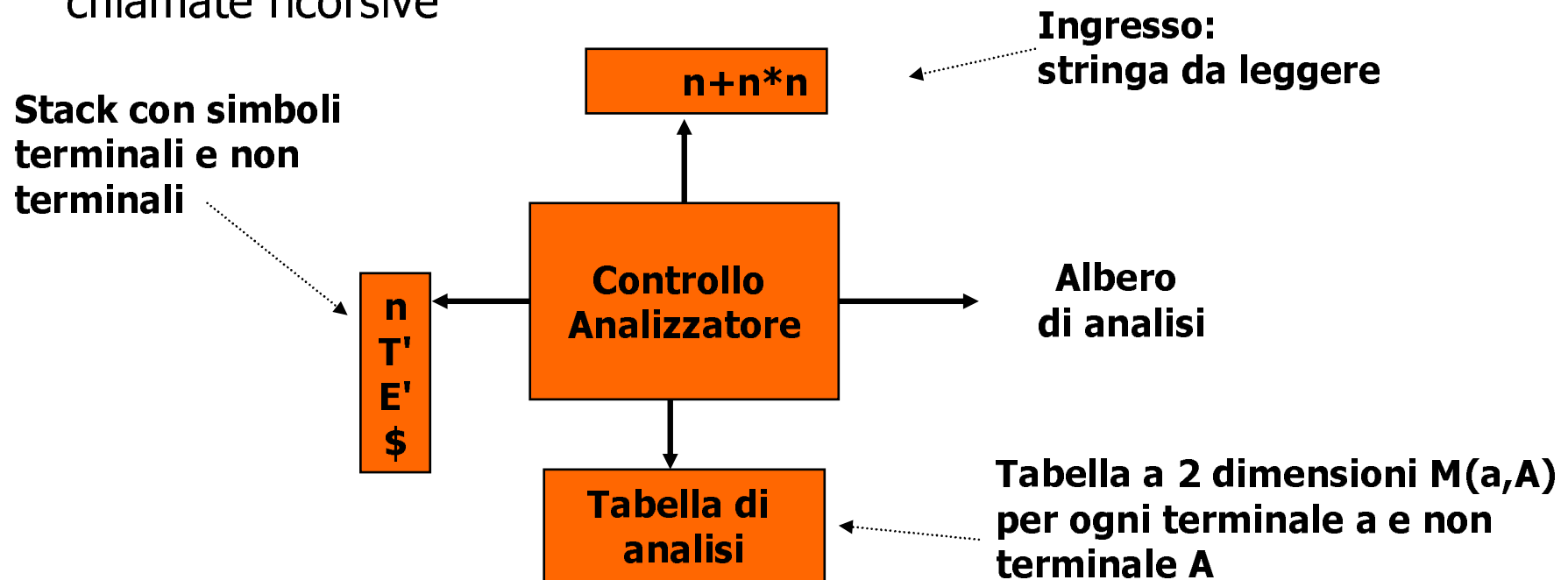
```
S -> if e then S  
S -> if e then S else S  
S -> i
```



```
S -> if e then S E | i  
E -> else S | ε
```

# Costruire un analizzatore predittivo

- ⌘ Riassumendo occorre: Eliminare eventuali ambiguità, eliminare la ricorsione a sinistra, fattorizzare la grammatica
- ⌘ Poi, per realizzare un analizzatore predittivo si usa uno stack invece di chiamate ricorsive





# Un esempio

## Analisi di $n+n*n$

Stack	Input	Produzioni
\$E	$n+n*n\$$	
$\$E'T$	$n+n*n\$$	$E \rightarrow TE'$
$\$E'T'F$	$n+n*n\$$	$T \rightarrow FT'$
$\$E'T'n$	$n+n*n\$$	$F \rightarrow n$
$\$E'T'$	$+n*n\$$	
$\$E'$	$+n*n\$$	$T' \rightarrow \epsilon$
$\$E'T+$	$+n*n\$$	$E' \rightarrow +TE'$
$\$E'T$	$n*n\$$	

## Grammatica

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid n$

## Tabella di analisi

	$n$	$+$	$*$	$($	$)$	$\$$
$E$	$TE'$			$TE'$		
$E'$		$+TE'$			$\epsilon$	$\epsilon$
$T$	$FT'$			$FT'$		
$T'$		$\epsilon$	$*FT'$		$\epsilon$	$\epsilon$
$F'$	$n$			$(n)$		

# Un esempio II

## Analisi di $n+n*n$

Stack	Input	Produzioni
\$E'T'F	$n*n\$$	$T \rightarrow FT'$
\$E'T'n	$n*n\$$	$F \rightarrow n$
\$E'T'	$*n\$$	
\$E'T'F*	$*n\$$	$T \rightarrow *FT'$
\$E'T'F	$n\$$	
\$E'T'n	$n\$$	$F \rightarrow n$
\$E'T'	$\$$	
\$E'	$\$$	$T' \rightarrow \epsilon$
$\$$	$\$$	$E' \rightarrow \epsilon$

## Grammatica

$E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \epsilon$   
 $T \rightarrow FT'$   
 $T \rightarrow *FT' \mid \epsilon$   
 $F \rightarrow (E) \mid n$

## Tabella di analisi

	$n$	$+$	$*$	$($	$)$	$\$$
$E$	$TE'$			$TE'$		
$E'$		$+TE'$			$\epsilon$	$\epsilon$
$T$	$FT'$			$FT'$		
$T'$		$\epsilon$	$*FT'$		$\epsilon$	$\epsilon$
$F$	$n$			$(n)$		

# Costruzione della tabella di analisi

⌘ Si costruiscono due funzioni FIRST e FOLLOW

☒  $w \in (V \cup T)^*$  ,  $\text{FIRST}(w)$  = l'insieme dei simboli terminali  
che iniziano le stringhe derivate da  $w$   
(se  $w \Rightarrow^* abAXs \dots$  allora  $a \in \text{FIRST}(w)$  )

☒  $A \in V$  ,  $\text{FOLLOW}(A)$  = l'insieme dei simboli terminali che possono  
seguire  $A$  in stringhe derivate da  $S$   
(se  $S \Rightarrow^* bcAas \dots$  allora  $a \in \text{FOLLOW}(A)$  )

# Calcolare FIRST

## ⌘ **FIRST(X), $X \in (V \cup T)$**

- 1) se  $X$  è un terminale, allora  $\text{FIRST}(X) = \{X\}$
- 2) se  $X \rightarrow \varepsilon$  è una produzione, allora  $\varepsilon \in \text{FIRST}(X)$
- 3) se  $X \rightarrow Y_1 Y_2 \dots Y_n$  è una produzione,
  - ⊠ se  $a \in \text{FIRST}(Y_i)$  e per ogni  $1 \leq j < i$  vale  $\varepsilon \in \text{FIRST}(Y_j)$  allora  $a \in \text{FIRST}(X)$   
( $Y_1 Y_2 \dots Y_{i-1} \Rightarrow^* \varepsilon$ )
  - ⊠ se per ogni  $1 \leq j \leq n$  vale  $\varepsilon \in \text{FIRST}(Y_j)$  allora  $\varepsilon \in \text{FIRST}(X)$

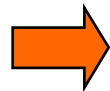
## ⌘ **FIRST(w), $w \in (V \cup T)^*$ , si calcola come in (3)**

- ⊠ Per costruire la tabella di analisi **occorre conoscere FIRST(w), per ogni regola  $A \rightarrow w$**

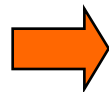
## ⌘ FIRST(X), viene calcolato ricorsivamente con le regole descritte sopra

# Calcolare FIRST: un esempio

$E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \varepsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid \varepsilon$   
 $F \rightarrow (E) \mid n$



$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, n \}$   
 $\text{FIRST}(E') = \{ +, \varepsilon \}$   
 $\text{FIRST}(T) = \text{FIRST}(F) = \{ (, n \}$   
 $\text{FIRST}(T') = \{ *, \varepsilon \}$   
 $\text{FIRST}(F) = \{ (, n \}$



$\text{FIRST}(TE') = \text{FIRST}(T) = \{ (, n \}$   
 $\text{FIRST}(+TE') = \{ + \}$     $\text{FIRST}(\varepsilon) = \{ \varepsilon \}$   
 $\text{FIRST}(FT') = \text{FIRST}(F) = \{ (, n \}$   
 $\text{FIRST}(*FT) = \{ * \}$     $\text{FIRST}(\varepsilon) = \{ \varepsilon \}$   
 $\text{FIRST}((E)) = \{ ( \}$     $\text{FIRST}(n) = \{ n \}$

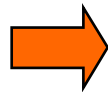
# Calcolare FIRST: un esempio II

$S \rightarrow ABCde \mid fAg$

$A \rightarrow ad \mid \epsilon$

$B \rightarrow b \mid \epsilon \mid C$

$C \rightarrow c \mid \epsilon \mid B$



$$\begin{aligned} \text{FIRST}(S) &= \{f\} \cup \text{FIRST}(A) \cup \text{FIRST}(B) \\ &\quad \cup \text{FIRST}(C) \cup \{d\} = \{f, a, b, c, d\} \end{aligned}$$

$\text{FIRST}(A) = \{a, \epsilon\}$

$\text{FIRST}(B) = \{b, \epsilon\} \cup \text{FIRST}(C) = \{b, c, \epsilon\}$

$\text{FIRST}(C) = \{c, \epsilon\} \cup \text{FIRST}(B) = \{b, c, \epsilon\}$

$$\begin{aligned} \text{FIRST}(ABCde) &= \text{FIRST}(A) \cup \text{FIRST}(B) \\ &\quad \cup \text{FIRST}(C) \cup \{d\} = \{a, b, c, d\} \end{aligned} \quad \text{FIRST}(fAg) = \{f\}$$

$\text{FIRST}(ad) = \{a\} \quad \text{FIRST}(\epsilon) = \{\epsilon\}$

$\text{FIRST}(b) = \{b\} \quad \text{FIRST}(\epsilon) = \{\epsilon\} \quad \text{FIRST}(C) = \{b, c, \epsilon\}$

$\text{FIRST}(c) = \{c\} \quad \text{FIRST}(\epsilon) = \{\epsilon\} \quad \text{FIRST}(B) = \{b, c, \epsilon\}$

# Calcolare FOLLOW

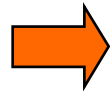
## ⌘ FOLLOW(X)

- ☒  $\$ \in \text{FOLLOW}(S)$ , dove  $\$$  rappresenta la fine della stringa
- ☒ se  $A \rightarrow \alpha B\beta$  è una produzione, allora  $\text{FIRST}(\beta) \setminus \{\epsilon\} \subseteq \text{FOLLOW}(B)$   
(B è seguito dai primi caratteri di  $\beta$ )
- ☒ se  $A \rightarrow \alpha B$  è una produzione oppure se  $A \rightarrow \alpha B\beta$  è una produzione e se  $\epsilon \in \text{FIRST}(\beta)$ , allora  $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$   
(se nella produzione non compare niente a destra di B allora B è seguito da ciò che segue A)

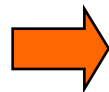
- ⌘ Prima occorre calcolare FIRST, poi FOLLOW viene calcolato applicando ricorsivamente le regole descritte sopra

# Calcolare FOLLOW: un esempio

$E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \varepsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid \varepsilon$   
 $F \rightarrow (E) \mid n$



$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, n \}$   
 $\text{FIRST}(E') = \{ +, \varepsilon \}$   
 $\text{FIRST}(T) = \text{FIRST}(F) = \{ (, n \}$   
 $\text{FIRST}(T') = \{ *, \varepsilon \}$   
 $\text{FIRST}(F) = \{ (, n \}$



$\text{FOLLOW}(E) = \{ \$ \} \cup \{ ) \} = \{ \$, ) \}$   
 $\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{ \$, ) \}$   
 $\text{FOLLOW}(T) = \text{FIRST}(E') \cup$   
 $\quad \text{FOLLOW}(E') \cup \text{FOLLOW}(E) = \{ +, \$, ) \}$   
 $\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{ +, \$, ) \}$   
 $\text{FOLLOW}(F) = \text{FIRST}(T') \cup \text{FOLLOW}(T) \cup$   
 $\quad \text{FOLLOW}(T') = \{ *, +, \$, ) \}$



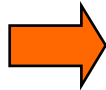
# Calcolare FOLLOW: un esempio II

$S \rightarrow ABCde \mid fAg$

$A \rightarrow ad \mid \epsilon$

$B \rightarrow b \mid \epsilon \mid C$

$C \rightarrow c \mid \epsilon \mid B$

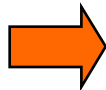


$\text{FIRST}(S) = \{f\} \cup \text{FIRST}(A) \cup \text{FIRST}(B)$   
 $\cup \text{FIRST}(C) \cup \{d\} = \{f, a, b, c, d\}$

$\text{FIRST}(A) = \{a, \epsilon\}$

$\text{FIRST}(B) = \{b, \epsilon\} \cup \text{FIRST}(C) = \{b, c, \epsilon\}$

$\text{FIRST}(C) = \{c, \epsilon\} \cup \text{FIRST}(B) = \{b, c, \epsilon\}$



$\text{FOLLOW}(S) = \{\$ \}$

$\text{FOLLOW}(A) = \text{FIRST}(B) \cup \text{FIRST}(C) \cup$   
 $\{d, g\} = \{b, c, d, g\}$

$\text{FOLLOW}(B) = \text{FIRST}(C) \cup \{d\} \cup \text{FOLLOW}(C) = \{b, c, d\}$

$\text{FOLLOW}(C) = \{d\} \cup \text{FOLLOW}(B) = \{b, c, d\}$

# Costruire la tabella di analisi

⌘ Per ogni produzione  $A \rightarrow w$  si **calcola FOLLOW(A) e FIRST(w)**

⌘ La tabella  $M[A,a]$  **è definita** da

☒ se  $A \rightarrow w$  e  $a \in \text{FIRST}(w)$ , allora  $M[A,a] = \{A \rightarrow w\}$

(se vedo  $a$  in ingresso e  $A$  è il prossimo simbolo sullo stack, allora espando  $A \rightarrow w$ )

☒ se  $A \rightarrow w$ ,  $\epsilon \in \text{FIRST}(w)$  e  $a \in \text{FOLLOW}(A)$ , allora  $M[A,a] = \{A \rightarrow \epsilon\}$

(se vedo  $a$  in ingresso e  $A$  è il prossimo simbolo sullo stack, allora espando  $A \rightarrow \epsilon$ )

# Costruire la tabella di analisi: un esempio

$\text{FIRST}(TE') = \text{FIRST}(T) = \{ (, n \}$   
 $\text{FIRST}(+TE') = \{ + \}$     $\text{FIRST}(\epsilon) = \{ \epsilon \}$   
 $\text{FIRST}(FT') = \text{FIRST}(F) = \{ (, n \}$   
 $\text{FIRST}(*FT') = \{ * \}$     $\text{FIRST}(\epsilon) = \{ \epsilon \}$   
 $\text{FIRST}((E)) = \{ ( \}$     $\text{FIRST}(n) = \{ n \}$

$E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \epsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid \epsilon$   
 $F \rightarrow (E) \mid n$

$\text{FOLLOW}(E) = \{ \$ \} \cup \{ ) \} = \{ \$, ) \}$   
 $\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{ \$, ) \}$   
 $\text{FOLLOW}(T) = \text{FIRST}(E') \cup$   
 $\quad \text{FOLLOW}(E') \cup \text{FOLLOW}(E) = \{ +, \$, ) \}$   
 $\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{ +, \$, ) \}$   
 $\text{FOLLOW}(F) = \text{FIRST}(T') \cup \text{FOLLOW}(T) \cup$   
 $\quad \text{FOLLOW}(T') = \{ *, +, \$, ) \}$

	n	+	*	(	)	\$
E	TE'			TE'		
E'		E → +TE'			ε	ε
T	FT'			FT'		
T'		ε	*FT'		ε	ε
F	n			(E)		

# Costruire la tabella di analisi: un esempio II

$\text{FIRST}(\text{ABCde}) = \{a, b, c, d\}$   $\text{FIRST}(\text{fAg}) = \{f\}$

$\text{FIRST}(\text{ad}) = \{a\}$   $\text{FIRST}(\epsilon) = \{\epsilon\}$

$\text{FIRST}(b) = \{b\}$   $\text{FIRST}(\epsilon) = \{\epsilon\}$   $\text{FIRST}(C) = \{b, c, \epsilon\}$

$\text{FIRST}(c) = \{c\}$   $\text{FIRST}(\epsilon) = \{\epsilon\}$   $\text{FIRST}(B) = \{b, c, \epsilon\}$

$S \rightarrow \text{ABCde} \mid \text{fAg}$

$A \rightarrow \text{ad} \mid \epsilon$

$B \rightarrow b \mid \epsilon \mid C$

$C \rightarrow c \mid \epsilon \mid B$

$\text{FOLLOW}(S) = \{\$ \}$

$\text{FOLLOW}(A) = \{b, c, d, g\}$

$\text{FOLLOW}(B) = \{b, c, d\}$

$\text{FOLLOW}(C) = \{b, c, d\}$

	a	b	c	d	e	f	g
S	ABCde	ABCde	ABCde	ABCde		fAg	
A	ad	$\epsilon$	$\epsilon$	$\epsilon$			$\epsilon$
B		b, C,	C,				
		$\epsilon$	$\epsilon$	$\epsilon$			
C		B	c, B				
		$\epsilon$	$\epsilon$	$\epsilon$			

**Più regole sono applicabili**

# Analizzatori discendenti e grammatiche LL(1)

⌘ Una grammatica la cui tabella di analisi per un analizzatore discendente non ha definizioni multiple si dice **LL(1)**

⊞ Left to right (scansione dell'ingresso)

⊞ leftmost (si espande il simbolo più a sinistra)

⊞ 1 simbolo per la previsione

⌘ Non tutte le grammatiche sono LL(1)

⊞ es. grammatiche ambigue e non fattorizzate a sinistra **non sono LL(1)**

⌘ Occorre

⊞ Eliminare l'ambiguità

⊞ eliminare la ricorsione a sinistra

⊞ Fattorizzare a sinistra

⊞ ma, **non è detto che si ottenga una grammatica LL(1)**

# **Analisi ascendente**

# Analisi ascendente (bottom up)

⌘ Si costruisce l'albero di analisi **dalle foglie verso la radice**.

☒ Si riduce una stringa allo scopo S

☒ Ad ogni passo una sottostringa che corrisponde alla **parte di sinistra** di una produzione è rimpiazzata col **simbolo di destra**

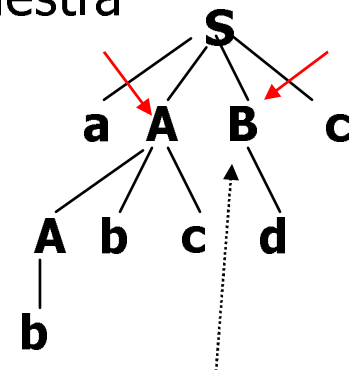
⌘ Se ci sono più stringhe da ridurre, si sceglie quella **più a sinistra**

☒ cioè si sceglie di espandere il simbolo terminale più a destra

$S \rightarrow aABe$   
 $A \rightarrow Abc|b$   
 $B \rightarrow d$

$abbcde \rightarrow aAbcde \rightarrow aAde \rightarrow aABe \rightarrow S$

Prima  $b \rightarrow A$  o  $d \rightarrow B$  ?  
La stringa più a sinistra



Prima  $A \rightarrow Abc$  o  $B \rightarrow d$  ?  
Il simbolo più a destra

# Le sottostringe "Handle" (di aggancio)

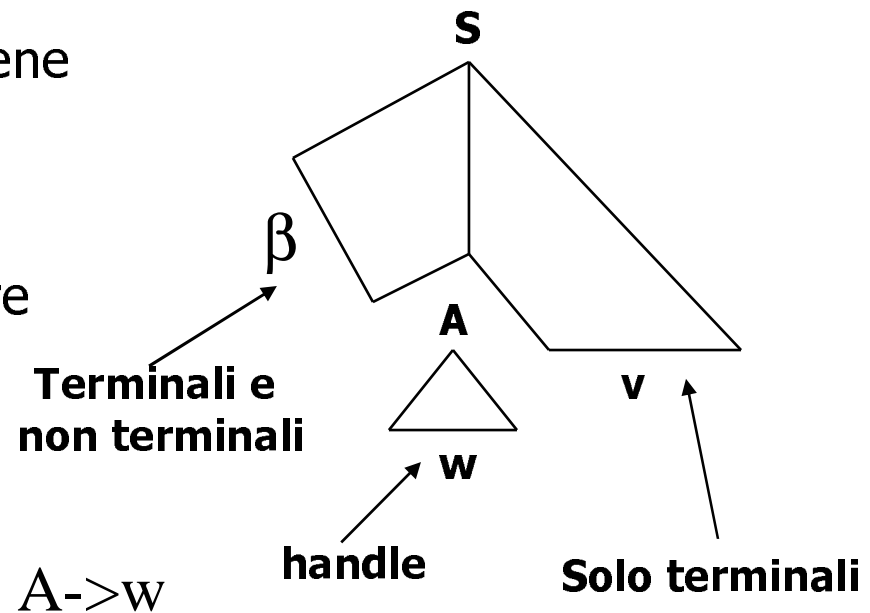
⌘ **Handle**: sottostringa tale che

☐ corrisponde la lato destro di una produzione  $A \rightarrow w$

☐ si può rimpiazzare nella stringa di partenza  $w$  con  $A$  ottenendo una derivazione destra

☐  $S \Rightarrow^* \beta A v \rightarrow \beta w v$ , dove  $v$  contiene **solo simboli terminali**

⌘ una grammatica **ambigua** può avere più handle





# Implementazione

⌘ Si usa uno **stack** per i simboli della grammatica

☒ Il parser inserisce simboli sullo stack fino a quando non trova un handle  $w$  (ad es.  $A \rightarrow w$ )

☒ Il parser riduce  $w$  al simbolo non terminale corrispondente  $A$

⌘ Il parser può fare le seguenti azioni

☒ **sposta (shift)**: il prossimo simbolo in ingresso è messo sullo stack

☒ **riduce (reduce)**: viene riconosciuto un handle sullo stack e sostituito da un non terminale

☒ **accetta**: sullo stack è rimasto  $S$  e in ingresso non c'è niente

☒ **errore**: si è riconosciuto un errore di sintassi

# Un esempio

Stack	Ingresso	Azione
\$	n+n*n\$	Sposta
\$n	+n*n\$	Riduci $E \rightarrow n$
\$E	+n*n\$	Sposta
\$E+	n*n\$	Sposta
\$E+n	*n\$	Riduci $E \rightarrow n$
\$E+E	*n\$	Sposta
\$E+E*	n\$	Sposta
\$E+E*n	\$	Riduci $E \rightarrow n$
\$E+E*nE	\$	Riduci $E \rightarrow E*nE$
\$E+E	\$	Riduci $E \rightarrow E+E$
\$E	\$	Accetta

$E \rightarrow E+E$

$E \rightarrow E*nE$

$E \rightarrow (E)$

$E \rightarrow n$

Si anche poteva scegliere di ridurre

# I conflitti



- ⌘ Ci sono grammatiche per le quali non si può applicare l'analisi ascendente perché noto il contenuto dello stack e il prossimo simbolo
  - ☒ è possibile sia ridurre che spostare (**conflitto shift/reduce**)
  - ☒ è possibile ridurre con più produzioni (**conflitto reduce/reduce**)
- ⌘ Le grammatiche per cui non esistono conflitti si dicono **LR**
- ⌘ Gli analizzatori bottom up
  - ☒ in caso di conflitto **usano un criterio** per decidere cosa fare (ad esempio, se esiste una precedenza fra gli operatori)
  - ☒ usano **grammatiche LR** (analizzatori LR)

# Analizzatori LR



## ⌘ Analisi LR(K)

- ☒ L (left to right): analisi della stringa da sinistra a destra
- ☒ R (rightmost derivation): si applica una derivazione che espande il non terminale più a destra
- ☒ K: numero di simboli di previsione

⌘ LR indica che la grammatica è LR(k) per un qualche k

# Funzionamento di un analizzatore LR

⌘ Lo **stack** contiene una stringa del tipo  $s_0X_1s_1\ldots X_ms_m$ ,

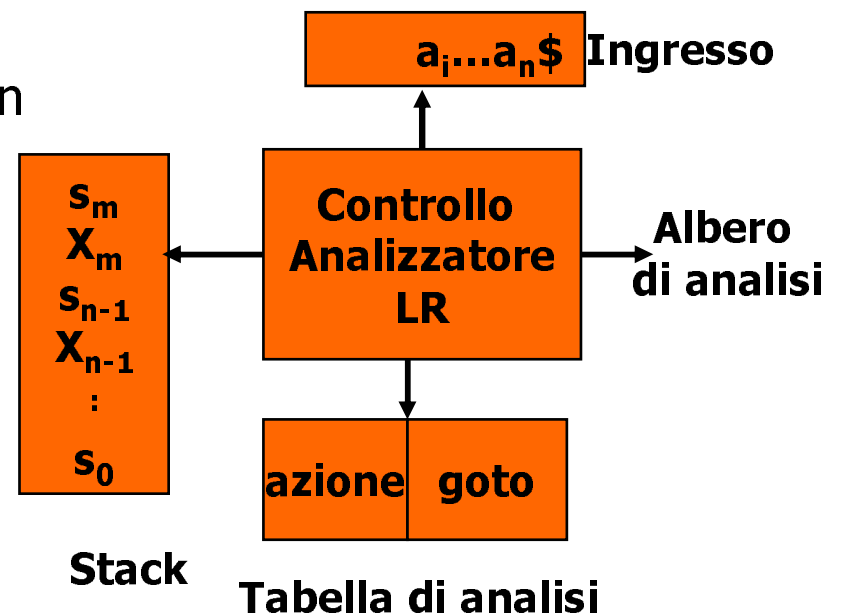
☒ Dove  $X_i$  è un **simbolo** della grammatica e  $S_i$  uno **stato**

☒ Ogni **stato**  $S_i$  riassume l'informazione contenuta nello stack al di sotto di esso

☒ Nell'implementazione i simboli  $X_i$  non sono necessari

⌘ L'azione del parser è determinata **dallo stato in testa allo stack e dal simbolo corrente**

⌘ La tabella di analisi è divisa in una **parte azione e una parte goto**



# Funzionamento di un analizzatore LR II

- ⌘ La **configurazione** dell'analizzatore è dato dal contenuto dello stack e dalla parte di ingresso ancora da leggere

$$s_0 X_1 s_1 \dots X_m s_m a_i \dots a_n$$

- ⌘ La configurazione rappresenta una stringa derivata sostituendo a destra

$$X_1 \dots X_m a_i \dots a_n$$

- ⌘ **Controllo dell'analizzatore:**

⏏ dati  $s_m$  stato in testa allo stack,  $a_i$  simbolo corrente si determina l'azione sulla base di  $AZIONE[s_m, a_i]$

# Funzionamento di un analizzatore LR III

## ⌘ **AZIONE**[ $s_m, a_i$ ] = **sposta s**

☒ Si spostano  $a_i$  e  $s$  sullo stack (si va allo stato  $s$ )



☒ Stack =  $s_0 X_1 s_1 \dots X_m s_m$

Ingresso =  $a_i$

$a_{i-1} \dots a_n$

☒ Stack =  $s_0 X_1 s_1 \dots X_m s_m a_i s$

Ingresso =

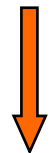
$a_{i-1} \dots a_n$

## ⌘ **AZIONE**[ $s_m, a_i$ ] = **riduci A → w**

( $w = X_{m-r+1} \dots X_m$ )

☒ Si rimuove  $r = |w|$  simboli dallo stack,

si mette  $A$  sullo stack  $a_i$  e si va allo stato  $s = \text{GOTO}(S_{m-r}, A)$



☒ Stack =  $s_0 X_1 s_1 \dots X_{m-r} s_{m-r} X_{m-r+1} s_{m-r+1} \dots X_m s_m$

Ingresso =  $a_i \dots a_n$

☒ Stack =  $s_0 X_1 s_1 \dots X_{m-r} s_{m-r} A s$

Ingresso =  $a_i \dots a_n$

## ⌘ **AZIONE**[ $s_m, a_i$ ] = **accetta**: L'analisi è completata

## ⌘ **AZIONE**[ $s_m, a_i$ ] = **errore**: Si è scoperto un errore

# Un esempio

stat	n	+	*	(	)	\$	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			

- |                           |                       |
|---------------------------|-----------------------|
| (1) $E \rightarrow E+T$   | (2) $E \rightarrow T$ |
| (3) $T \rightarrow T * F$ | (4) $T \rightarrow F$ |
| (5) $F \rightarrow (E)$   | (6) $F \rightarrow n$ |

Stack	Input	Azione	Goto
0	n*n+n\$	s5	
0n5	*n+n\$	r6	3
0F3	*n+n\$	r4	2
0T2	*n+n\$	s7	
0T2*7	n+n\$	s5	
0T2*7n5	+n	r6	10
0T2*7F10	+n\$	r3	2



# Un esempio II

stat	n	+	*	(	)	\$	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			

(1)  $E \rightarrow E+T$

(2)  $E \rightarrow T$

(3)  $T \rightarrow T * F$

(4)  $T \rightarrow F$

(5)  $F \rightarrow (E)$

(6)  $F \rightarrow n$

Stack	Input	Azione	Goto
0T2	+n\$	r2	1
0E1	+n\$	s6	
0E1+6	n\$	s5	
0E1+6n5	\$	r6	3
0E1+6F3	\$	r4	9
0E1+6T9	\$	r1	1
0E1	\$	acc	

# Perché gli analizzatore LR funzionano?

⌘ La stringa  $\beta w$  è un **prefisso riducibile** di  $\beta w v$

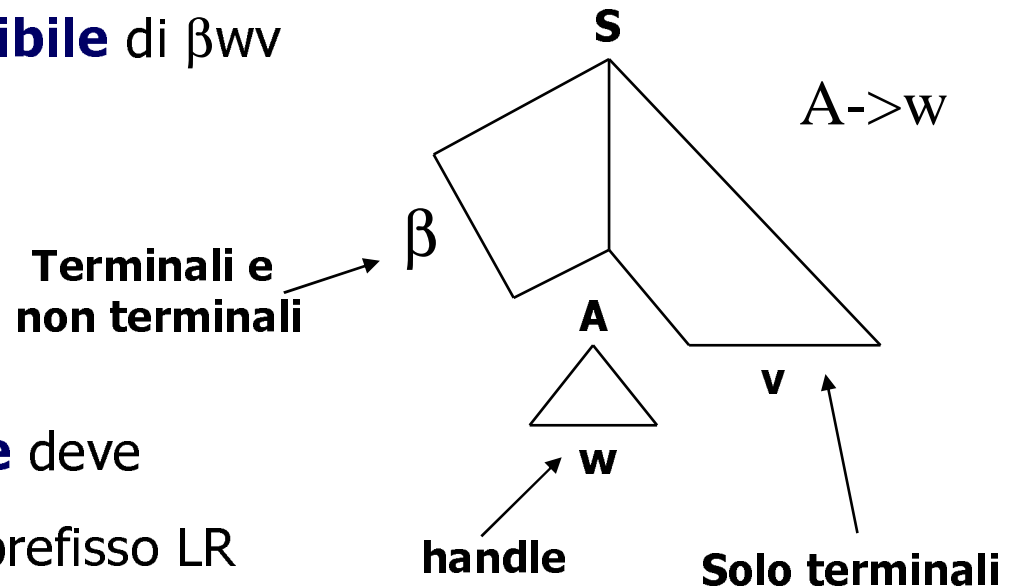
⌘ Si dice **prefisso LR** un qualsiasi prefisso di un prefisso riducibile

⌘ Un analizzatore LR per **funzionare** deve

☒ riconoscere la presenza di un prefisso LR

☒ decidere se il prefisso è riducibile

☒ decidere con quale regola ridurre



# Perché gli analizzatori LR funzionano? II

- ⌘ I simboli dello stack sono **prefissi LR**
  - ☒ sono prefissi riducibili quando si fa una riduzione
  - ☒ sono semplici prefissi LR quando si fa uno spostamento
- ⌘ Si può dimostrare che i prefissi LR sono **riconoscibili da un automa a stati finiti**
- ⌘ Lo stato in cima allo stack è rappresentato dallo stato in cui si trova l'automato a stati finiti
- ⌘ Un automato LR **utilizza lo stato in testa allo stack per sapere a che punto è nella lettura del prefisso e si aiuta con il primo simbolo in ingresso per decidere cosa fare**

# Costruzione dell'automa che riconosce i prefissi

⌘ Un **elemento LR(0)** di una grammatica è una produzione annotata con un punto

☐  $A \rightarrow .xya$

☐  $A \rightarrow x.ya$

☐  $A \rightarrow xy.a$

☐  $A \rightarrow xya.$

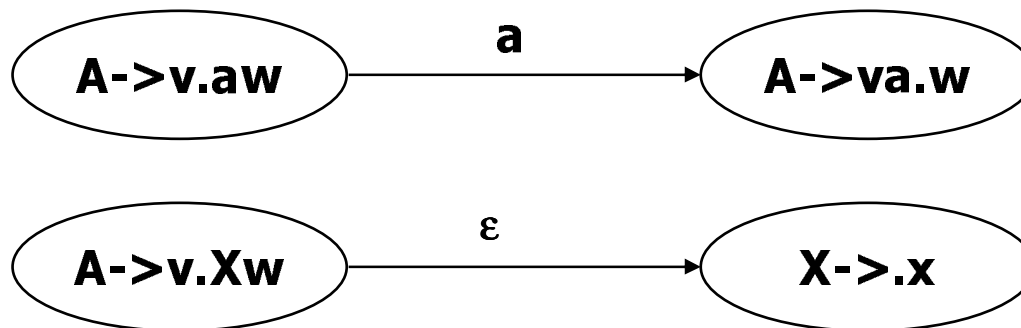
⌘ Un elemento è individuato da una coppia di indici: numero produzione, posizione punto

⌘ Il **punto** tiene traccia di quanto è stato già letto

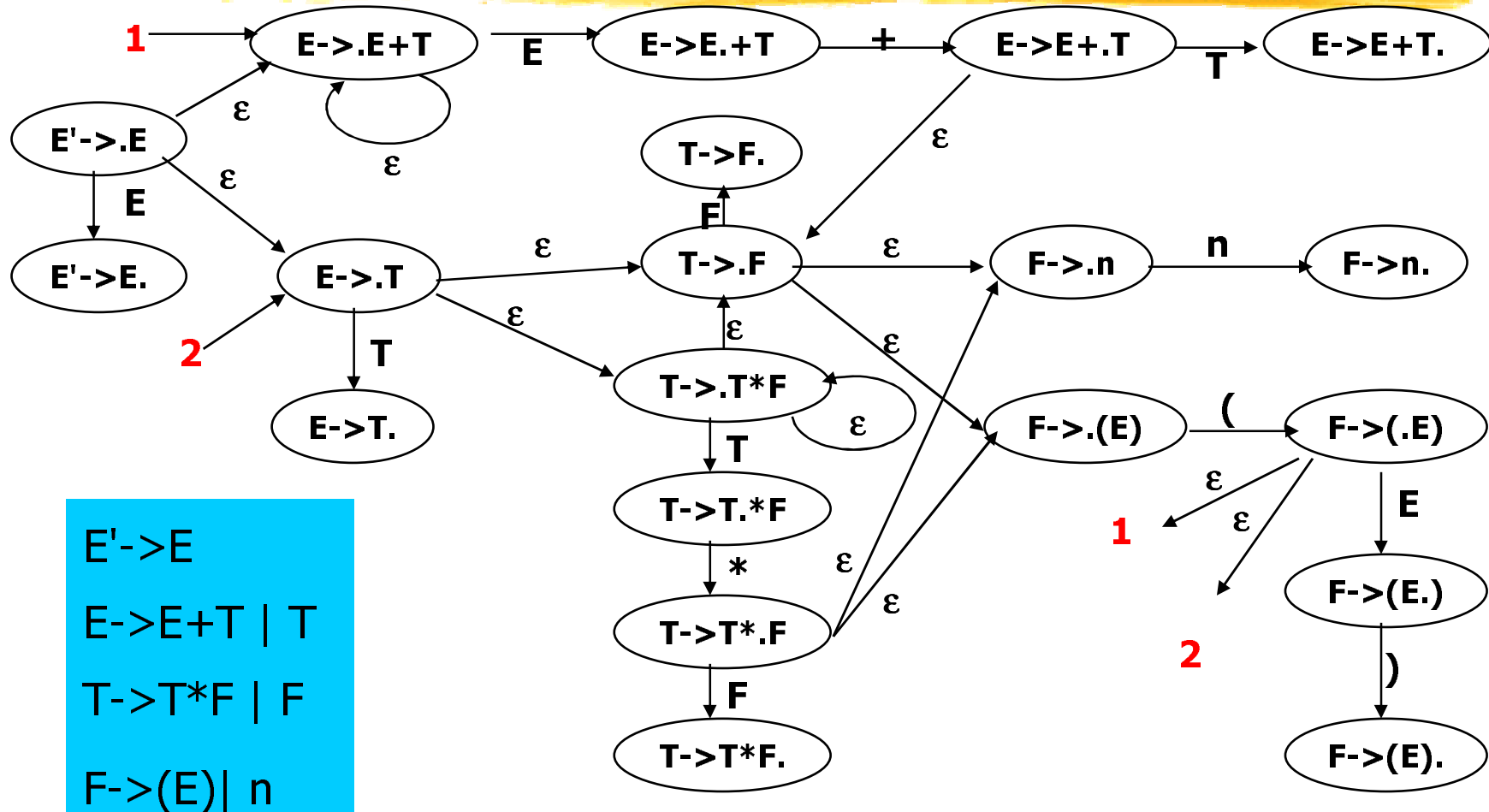
# Costruzione dell'automa che riconosce i prefissi II

## ⌘ L'automa

- ☒ gli **stati** sono elementi LR(0)
- ☒ se  $A \rightarrow vaw \in P$  e  $a \in T$ , c'è un **arco**  
 $[A \rightarrow v.aw] \xrightarrow{a} [A \rightarrow va.w]$
- ☒ se  $A \rightarrow vXw \in P$  e  $X \in V$  c'è un **arco**  
 $[A \rightarrow v.Xw] \xrightarrow{\varepsilon} [X \rightarrow .x]$



# Costruzione dell'automa che riconosce i prefissi: un esempio



# L'automa deterministico

⌘ Gli automi costruiti con il metodo precedente sono **non deterministici**

☒ occorre trasformati in automi deterministici

☒ si usa la solita procedura per passare da NFA a DFA

⌘ Alla fine **ogni stato**  $I$  dell'automa deterministico corrisponde ad **un insieme di elementi LR(0)**

☒ ad es.  $I = \{F \rightarrow \cdot n, F \rightarrow \cdot (E), T \rightarrow T^* \cdot F\}$

# L'automata deterministico: un esempio

$I_0 = \{E' \rightarrow .E, E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .F, \\ T \rightarrow .T*F, F \rightarrow .n, F \rightarrow .(E)\}$

$I_1 = \{E' \rightarrow E., E \rightarrow E.+T\}$

$I_2 = \{E \rightarrow T., E \rightarrow E.+T\}$       $I_3 = \{T \rightarrow F.\}$

$I_4 = \{F \rightarrow (.E), E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .F, \\ T \rightarrow .T*F, F \rightarrow .n, F \rightarrow .(E)\}$

$I_5 = \{F \rightarrow n.\}$

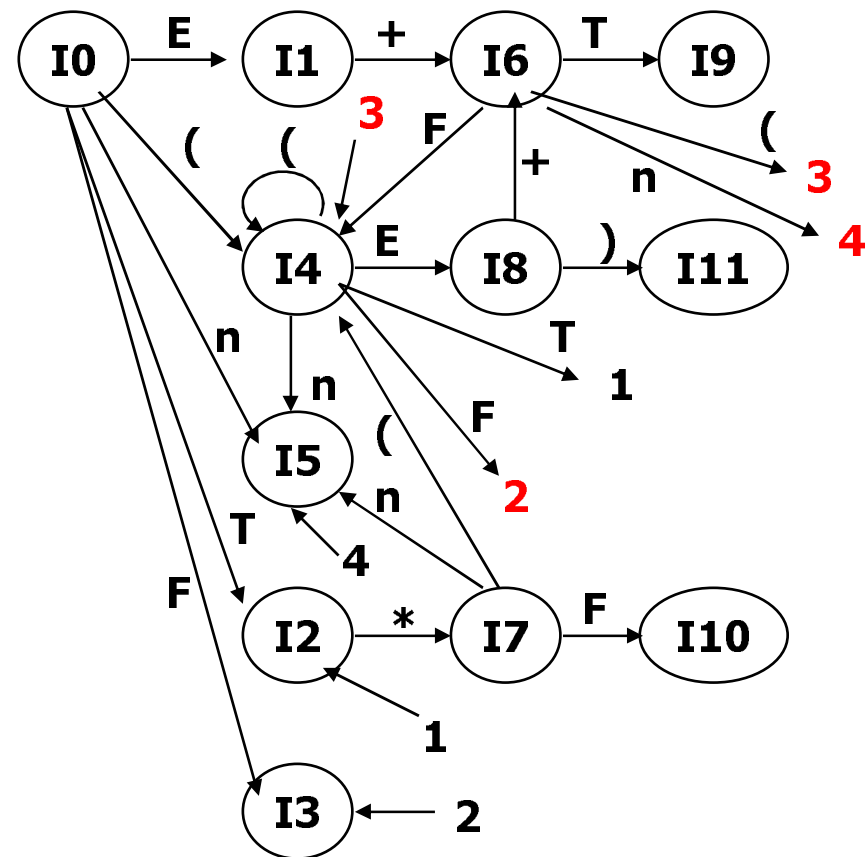
$I_6 = \{T \rightarrow .F, T \rightarrow .T*F, F \rightarrow .n, F \rightarrow .(E)\}$

$I_7 = \{F \rightarrow .n, F \rightarrow .(E), T \rightarrow T*.F\}$

$I_8 = \{F \rightarrow (E.), F \rightarrow E.+T\}$

$I_9 = \{E \rightarrow E+T., T \rightarrow T.*F\}$

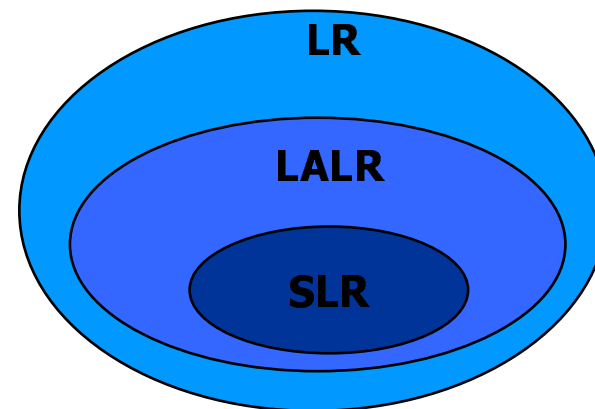
$I_{10} = \{T \rightarrow T*F.\}$       $I_{11} = \{F \rightarrow (E).\}$





# Costruzione di tabelle di analisi

- ⌘ Dopo l'automa con stati  $\{I_1, \dots, I_n\}$  si costruiscono le **tabelle di analisi**
- ⌘ Esistono vari modi di costruire le tabelle
  - ☒ Analizzatori **SLR** (simple LR), **LALR** (Lookahead LR), e **LR**
  - ☒ **tabelle più grandi** e complesse da costruire corrispondono a minori conflitti e **linguaggi più generici**
  - ☒ **tabelle più piccole** e semplici da costruire corrispondono a più conflitti e **linguaggi più ristretti**
- ⌘ La maggior parte dei compilatori usa LALR
- ⌘ **Noi vedremo SLR(1)**



# Costruzione di tabelle di analisi SLR

⌘ Per un analizzatore **SLR (simple LR)**, AZIONE e GOTO saranno

## ⌘ Spostamenti

☞ se  $a \in T$  e l'automa contiene la transizione  $I_h \xrightarrow{a} I_k$

AZIONE[h,a]= sposta k

☞ (si passa al nuovo stato dell'automa, il prefisso **non è ancora riducibile**)

## ⌘ Riduzioni

⌘ se  $[A \rightarrow w.] \in I_h$  e  $a \in \text{FOLLOW}(A)$

AZIONE[h,a]= riduci  $A \rightarrow w$

☞ (Si riduce, il prefisso **è riducibile**)

# Costruzione di tabelle di analisi SLR II

## ⌘ Riduzioni

⌘ se  $[S' \rightarrow S.] \in I_h$

AZIONE[h,\$] = accetta

⌘  $(S' \rightarrow S$  è una **nuova regola** aggiunta al linguaggio)

## ⌘ La tabella GOTO

⌘ se  $I_h \xrightarrow{A} I_k$

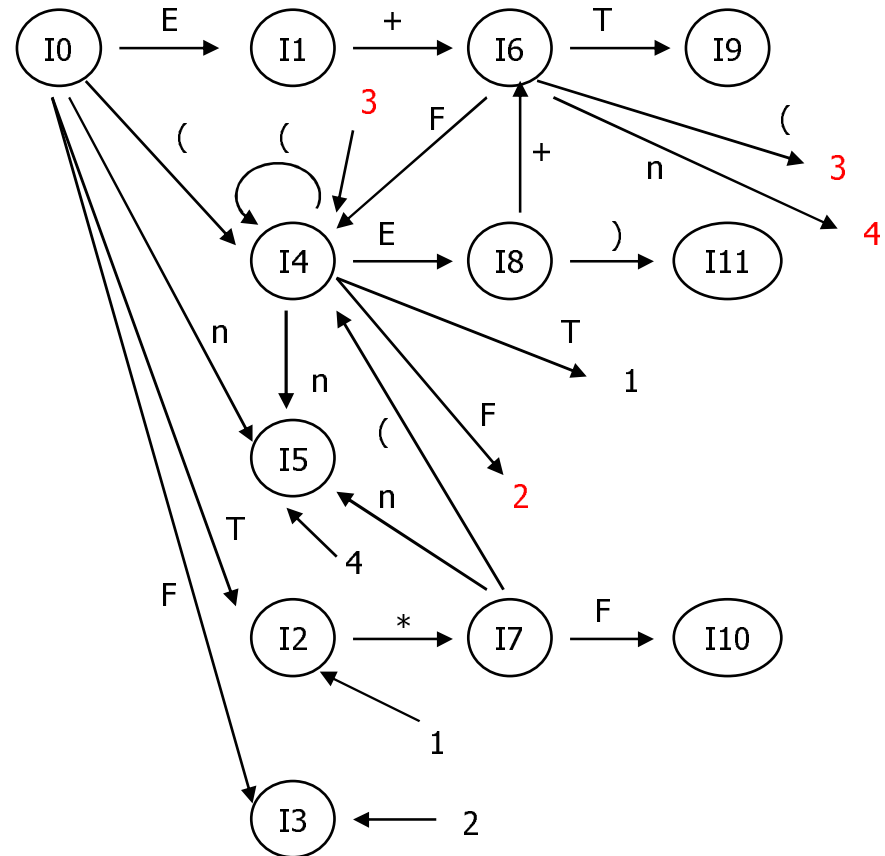
GOTO[h,A] = k

⌘ Lo stato iniziale è quello che contiene  $[S' \rightarrow .S]$

⌘ Le componenti non definite producono un errore

# AZIONE: gli spostamenti

state	n	+	*	(	)	\$
0	s5			s4		
1		s6				
2			s7			
3						
4	s5			s4		
5						
6	s5			s4		
7	s5			s4		
8		s6			s11	
9			s7			
10						
11						



# Costruzione della tabella

## AZIONE: le riduzioni

stato	n	+	*	(	)	\$
0	s5			s4		
1		s6				acc
2		r2	s7		r2	r2
3		r4	r4		r4	r4
4	s5			s4		
5		r6	r6		r6	r6
6	s5			s4		
7	s5			s4		
8		s6			s11	
9		r1	s7		r1	r1
10		r3	r3		r3	r3
11		r5	r5		r5	r5

(0)  $E' \rightarrow E$   
 (1)  $E \rightarrow E+T$   
 (2)  $E \rightarrow T$   
 (3)  $T \rightarrow T*F$   
 (4)  $T \rightarrow F$   
 (5)  $F \rightarrow (E)$   
 (6)  $F \rightarrow n$

$I1 = \{E' \rightarrow E., \dots\}$   
 $I2 = \{E \rightarrow T., \dots\}$   
 $I3 = \{T \rightarrow F., \dots\}$   
 $I5 = \{F \rightarrow n., \dots\}$   
 $I9 = \{E \rightarrow E+T., \dots\}$   
 $I10 = \{T \rightarrow T*F., \dots\}$   
 $I11 = \{F \rightarrow (E)., \dots\}$

$FOLLOW(E') = \{\$, \}$   $FOLLOW(E) = \{+, ), \$\}$

$FOLLOW(T) = \{+, *, ), \$\}$   $FOLLOW(F) = \{+, *, ), \$\}$

Age Group	Percentage
18-24	~10%
25-34	~35%
35-44	~25%
45-54	~15%
55-64	~10%
65-74	~5%
75-84	~2%
85+	~1%

Fondamenti II 2003



# Grammatiche LR(1)

- ⌘ **Una grammatica si dice SLR(1) se** la corrispondente tabella di analisi costruita come appena mostrato **non contiene conflitti**
- ⌘ Costruzione di tabelle per analizzatori LR (idea base)
  - ⏏ Gli stati memorizzano **esplicitamente i non terminali b che possono seguire un handle y**
  - ⏏ Gli elementi di una grammatica LR(1) sono del tipo  $[A \rightarrow .xya, b]$ ,  $[A \rightarrow x.ya, b]$
  - ⏏ Si costruisce un automa deterministico con tali elementi
  - ⏏ Con  $[A \rightarrow w., b]$  **si riduce solo se il prossimo simbolo è b**

# YACC: Yet Another Compiler Compiler

- ⌘ **Yacc** è un generatore di analizzatori sintattici in linguaggio C a partire dalla grammatica LALR(1)
- ⌘ L'ingresso è un file che descrive la grammatica: **simboli non terminali, simboli terminali, produzioni**

**Terminali** →

**Non terminali** →

**Produzioni** →

```
%token <val> NUM
%token <sptr> VAR
%type <val> exp

%%
input: /* vuota */
      | input exp '\n'
;
exp:  NUM { $$=$1; }
     | VAR { $$=$1->value.var; }
     | VAR '=' exp { $$=$3;$1->value.var=$3; }
     | exp '+' exp { $$=$1+$3; }
     | exp '*' exp { $$=$1*$3; }
     | '(' exp ')' { $$=$2; }
;
```



# I simboli

- ⌘ Ad ogni simbolo è associato **un tipo e un valore semantico**
- ⌘ Il valore semantico definisce **il significato di un simbolo**
  - ☒ Un intero NUM: la costante associata al numero
  - ☒ Una variabile VAR: il puntatore alla locazione della variabile
  - ☒ Un'espressione expr: il valore dell'espressione
- ⌘ Il tipo definisce **il tipo C** del valore associato al valore semantico

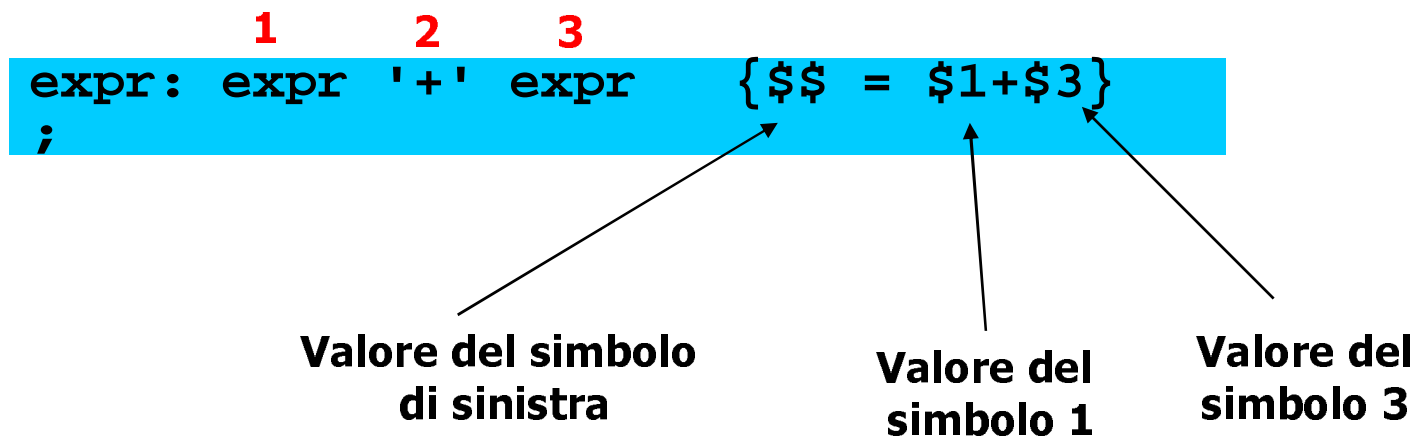
```
%union{
    double val;
    symtbl *sptr;
}
%token <val> NUM
%token <sptr> VAR
%type <val> exp
. . . . .
```

Il tipo di espressioni e numeri

Il tipo delle variabili è un puntatore alla tabella dei simboli

# Le produzioni

- ⌘ Alle produzioni **sono associate azioni** (codice C) che ne definisco il significato
- ⌘ Ogni volta che viene applicata la regola **si esegue il codice**
- ⌘ L'azione permette di combinare i valori associati ai simboli per **calcolare il valore semantico del simbolo a sinistra** della produzione



# Una calcolatrice

**Definisce la  
precedenza fra  
gli operatori**

**Calcola il valore  
di un'espressione**

```
%union{
    double val;
    symbtbl *sptr;}
%token <val> NUM
%token <sptr> VAR
%type <val> exp

%right '='
%left '+'
%left '*'

%%
input: /* vuota */
      | input exp '\n'
;

exp:  NUM {$$=$1;}
     | VAR {$$=$1->value.var;}
     | VAR '=' exp {$$=$3;$1->value.var=$3;}
     | exp '+' exp {$$=$1+$3;}
     | exp '*' exp {$$=$1*$3;}
     | '(' exp ')' {$$=$2;}
;
```

# YACC e LEX

## ⌘ Il valore semantico

☒ nella calcolatrice è il valore dell'espressione

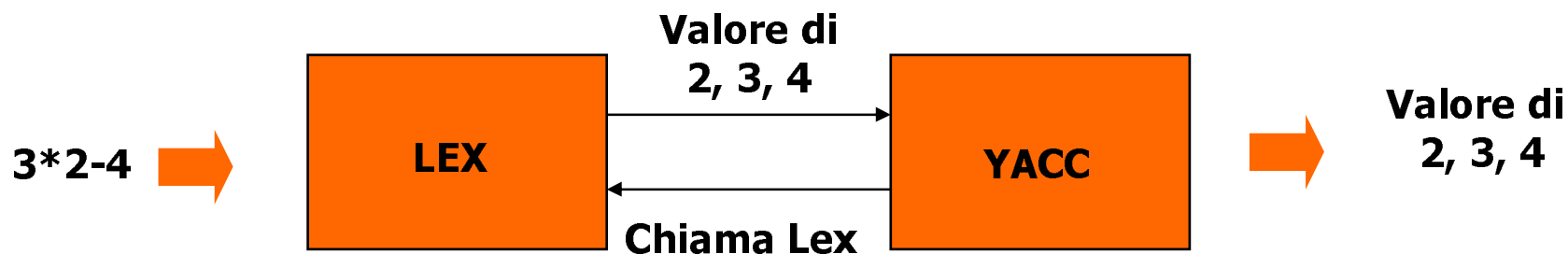
☒ in compilatore è un albero sintattico

## ⌘ Il valore dei simboli

☒ **simboli terminali** viene calcolato da **LEX**

☒ **simboli non terminali** viene calcolato da **YACC**

⌘ YACC chiama automaticamente LEX attraverso una funzione (yyparse()) per fare l'analisi lessicale



# **Grammatiche non contestuali**

# Grammatiche formali

- ⌘ Esistono altre grammatiche oltre quelle libere da contesto
- ⌘ Il tipo **più generale di grammatiche** prevede regole nel quale la parte sinistra può essere una **qualsiasi stringa di terminali e non terminali**
- ⌘ Una **grammatica formale** è una quadrupla  $G=(V,T,P,S)$ 
  - ⌘  $V$  è un insieme di simboli non terminali,  $T$  un insieme di simboli terminali,  $S \in V$  è lo scopo o simbolo di partenza
  - ⌘  $P$  un insieme di **produzioni** del tipo
    - ⌘  **$v \rightarrow w$  , dove  $v, w \in (V \cup T)^*$**

# Grammatiche formali: un esempio

⌘ Le seguente grammatica genera le stringhe del tipo  $a^n b^n$

$T=\{a,b\}, V=\{A,B,S\}$

(1)  $S \rightarrow ABS$

(2)  $S \rightarrow \epsilon$

(3)  $BA \rightarrow AB$

(4)  $BS \rightarrow b$

(5)  $Bb \rightarrow bb$

(6)  $Ab \rightarrow ab$

(7)  $Aa \rightarrow aa$

## Derivazione di aaabbb

$S \xrightarrow{(1)} AB \quad S \xrightarrow{(1)} ABAB \quad S \xrightarrow{(1)} ABABAB$   
 $\xrightarrow{(1)} ABA \quad BA \quad BS \xrightarrow{(3)} A \quad BA \quad ABBS \xrightarrow{(3)} AA \quad BA \quad ABBS \xrightarrow{(3)} AAA \quad BA \quad BBS \xrightarrow{(3)} AAABB \quad BS \xrightarrow{(4)} AAAB \quad Bb \xrightarrow{(5)} AAA \quad Bb \quad b \xrightarrow{(5)} AA \quad Ab \quad bb \xrightarrow{(7)} A \quad Aa \quad bbb \xrightarrow{(7)} Aa \quad abbb \xrightarrow{(7)} aaabbb$

### Il trucco:

- A e B possono essere scambiati
- gli A e B sono convertiti in a e b partendo dal fondo si propaga al resto della stringa

# Grammatiche formali II

## ⌘ Grammatiche generiche

☒ nessuna limitazione

## ⌘ Grammatiche contestuali (def. 1 e def. 2 sono equivalenti, esiste un algoritmo che trasforma una grammatica da una a l'altra forma)

☒ **def. 1:** le produzioni sostituiscono un solo non simbolo  $A$ ;

$vAw \rightarrow v\alpha w$ , dove  $A \in V$ ,  $v, w \in (V \cup T)^*$  e  $\alpha \in (V \cup T)^+$

$v, w$  **definiscono il contesto** nel quale la sostituzione è ammissibile

☒ **def. 2: non** sono ammesse produzioni che realizzano **contrazioni**

$v \rightarrow w$ , dove  $v, w \in (V \cup T)^*$  e  $|v| \leq |w|$



# Grammatiche contestuali e non contestuali

- ⌘ Le grammatiche contestuali hanno **un maggior potere espressivo** delle non contestuali
- ⌘ Linguaggi che appartengono alla classe dei linguaggi contestuali ma non a quella dei linguaggi non contestuali

⌘ il linguaggio degli **identificatori**:

Le stringhe  $wcw$ , dove  $v, w \in T^*$   $c \in T$

si verifica che un'occorrenza di un identificatore (seconda occorrenza di  $w$ ) sia uguale a quello dichiarato (prima occorrenza di  $w$ )

⌘ il linguaggio delle **chiamate di funzioni**

Le stringhe  $a^n b^m c^n d^m$ , dove  $a, b, c, d \in T$

si verifica che il tipo dei parametri di un'occorrenza di procedura (cccdd indica 3 parametri interi e 2 reali) corrispondano a quelli dichiarati (aaabb)

# Grammatiche contestuali: un esempio

⌘ Le seguente grammatica genera le stringhe del tipo  $wcw$  con  $w \in (a|b)^*$

(1) $S \rightarrow AA'S$	(11) $Ac \rightarrow ac$
(2) $S \rightarrow BB'S$	(12) $Bc \rightarrow bc$
(3) $S \rightarrow C$	(13) $Aa \rightarrow aa$
(4) $A'A \rightarrow AA'$	(14) $Ab \rightarrow ab$
(5) $B'B \rightarrow BB'$	(15) $Ba \rightarrow ba$
(6) $A'B \rightarrow BA'$	(16) $Bb \rightarrow bb$
(7) $B'A \rightarrow AB'$	(17) $cA' = ca$
(8) $A'C \rightarrow CA'$	(18) $cB' = cb$
(9) $B'C \rightarrow CB'$	(19) $aA' = aa$
(10) $C \rightarrow c$	(20) $aB' = cb$
	(21) $bA' = ba$
	(22) $bB' = bb$

## Derivazione di $abcbab$

$S \xrightarrow{(1)} AA' S \xrightarrow{(2)} AA'BB' S \xrightarrow{(3)}$   
 $\xrightarrow{(3)} A A'B B'C \xrightarrow{(6)} ABA' B'C \xrightarrow{(9)}$   
 $\xrightarrow{(9)} AB A'C B' \xrightarrow{(8)} AB C A'B' \xrightarrow{(10)}$   
 $\xrightarrow{(10)} A Bc A'B' \xrightarrow{(12)} Ab cA'B' \xrightarrow{(11)}$   
 $\xrightarrow{(11)} ab cA' B' \xrightarrow{(17)} abc aB' \xrightarrow{(20)}$   
 $\xrightarrow{(20)} abcbab$

## Il trucco:

- **A e B sono portati in testa alla stringa, A' e B' in fondo, C al centro**
- **L'ordine fra A e B e fra A' e B' viene mantenuto**
- **la conversione di A, B, A', B', C in a, b, c inizia dal centro e si propaga al resto della stringa**

# La gerarchia di Chomsky

- ⌘ **Chomsky** ha chiarito quale sia il potere espressivo delle grammatiche regolari, con contestuali, contestuali senza restrizioni
- ⌘ Le inclusioni sono tutte **inclusioni strette**: ci sono linguaggi che generati dalle grammatiche senza restrizioni e non dalle grammatiche contestuali ...
- ⌘ I linguaggi **non contestuali e regolari**
  - ☒ si usano negli analizzatori sintattici e lessicali
  - ☒ sono facili da implementare
- ⌘ I **linguaggi contestuali**
  - ☒ difficili da implementare
  - ☒ lo sono i comuni linguaggi di programmazione
  - ☒ L'analisi semantica serve a riconoscere le caratteristiche non riconoscibili solo con gli analizzatori sintattici



# Cacolabilità e linguaggi

⌘ I **linguaggi contestuali** sono un sottoinsieme dei **linguaggi decidibili**

☒ Un linguaggio  $L$  è decidibile se esiste un programma (della macchina di Turing) che data una stringa  $w$  decide se appartiene del linguaggio  
**(risponde a " $w \in L$ ?")**

⌘ I linguaggi generati dalle **grammatiche senza restrizioni** sono tutti i linguaggi **ricorsivamente enumerabili**

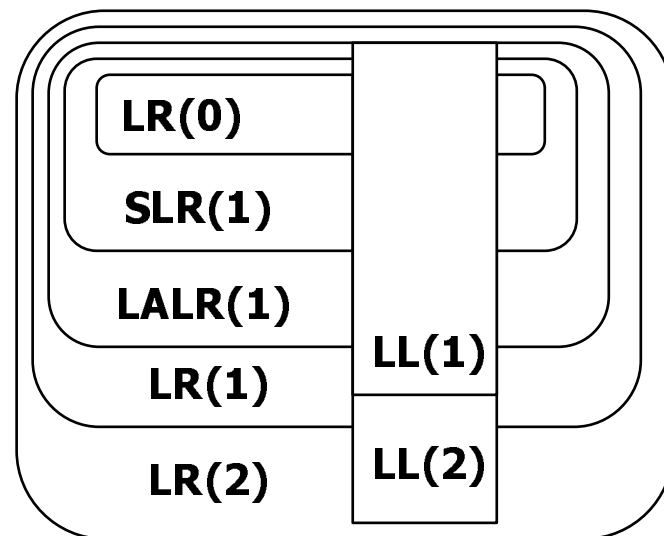
☒ Un linguaggio  $L$  è ricorsivamente enumerabile se esiste un programma che genera tutte le stringhe di  $L$   
**(rispondendo a " $w \in L$ ?" il programma può non terminare)**

# Grammatiche contestuali

⌘ Per le grammatiche valgono le seguenti relazioni

⊞  $\text{LR}(0) \subset \text{SLR}(1) \subset \text{LALR}(1) \subset \text{LR}(1) \subset \text{LR}(2)$

⊞  $\text{LL}(k) \subset \text{LR}(k)$ , per ogni  $k$  (per questo si preferisce le LR)



# Linguaggi contestuali

⌘ Un linguaggio è **LR(k)**,  
    ☒ **se esiste una grammatica LR(k) che lo genera**

⌘ Un linguaggio è  
    ☒ **LR se è LR(K) per qualche k**

⌘ Per i linguaggi vale  
    ☒ **LR(K)=LR(1)=LR(0)=**  
    **=SLR(1)=LALR(1)**

☒ **LL(1) ⊂ LL(2) ⊂ LL(3)**

⌘ Per la maggior parte dei linguaggi  
    è possibile trovare grammatiche  
    di ogni tipo

