


# Tutorato linguaggi formali e compilatori

Ettore Speziale

Politecnico di Milano

28 febbraio 2011


# Indice


# Sommario


# Contenuti



Il tutorato è organizzato per aiutarvi a preparare l'esame. Vedremo esercizi sia "teorici" che "pratici":

**teoria** espressioni regolari, grammatiche, ...

**pratica** modificare la macchina ACSE

In poche parole:

- risolveremo un tema d'esame


# Organizzazione



Per questa sessione vengono offerte due lezioni:

- una in Inglese
- una in Italiano

Altre lezioni saranno tenute in prossimità delle sessioni d'esame:

- una lezione a luglio
- due lezioni a settembre

Ogni lezione è suddivisa in due parti:

- 3 ore per la parte di teoria
- 3 ore per la parte di laboratorio


# Materiale



Queste slides possono essere scaricate dal mio sito:

`http://home.dei.polimi.it/speziale/`

Cercatele nella sezione didattica riferita all'anno corrente. Troverete diverse piccole figure:

- questo perché è difficile, per esempio, far entrare un AST in una slide
- comunque le immagini sono scalabili, potete ingrandirle quanto volete


# Sommario

# Progetto di espressioni regolari

Il linguaggio $L$ è composto dai caratteri $a, b$ con le seguenti restrizioni:

1. non contiene la stringa vuota
2. la sotto-stringa $ab$ occorre un numero pari di volte

Si richiede di:

1. scrivere tre frasi di lunghezza 6
2. scrivere l'espressione regolare $R$ generante $L$ con i soli operatori unione, concatenamento, stella e croce
3. verificare se vale la relazione $L(R) = L(R^*)$

# Rappresentare $L$

Il testo è scritto in linguaggio naturale, ma descrivere il linguaggio $L$ semi-formalmente può essere utile:

$$L(R) = \{x \in \Sigma^*_{a,b} | x \neq \epsilon \cap even(\#x_{ab})\}$$

Alcune persone ragionano meglio con quest'ultima rappresentazione.

# Stringhe d'esempio

Questo punto è molto importante:

- permette di capire come è fatto il linguaggio

Nello specifico, osservo che 0 è un numero pari quindi:

$$\{s_1, s_2\} = \{a^6, b^2a^4\} = \{aaaaaa, bbaaaa\} \subset L(R)$$

Il pari successivo è 2, quindi considero:

$$KabHabJ$$

Scelgo opportunamente $K, H, J$:

$$s_3 = subs(K \Rightarrow \epsilon, H \Rightarrow \epsilon, J \Rightarrow a^2) = (ab)^2a^2 = ababaa \in L(R)$$

# Espressione regolare I

Procediamo per passi; partiamo dal vicolo più stringente e da una espressione regolare base:

$$even(\#x_{ab}), R_0 = abab$$

Poi aggiungiamo caratteri in modo che le stringhe generate stiano sempre all'interno di $L$:

- tra le due $ab$ non devo far formare altre $ab$:

$$R_1 = ab^+a^+b$$

- ripeto $R_1$ un numero arbitrario, diverso da 0, di volte:

$$R_2 = (ab^+a^+b)^+$$

# Espressione regolare II

- considero i limiti del ciclo e li espando:

$$R_3 = (a^+ b^+ a^+ b^+)^+$$

- mi accorgo che la stringa potrebbe iniziare anche con delle $b$ e finire con delle $a$:

$$R_4 = b^*(a^+ b^+ a^+ b^+)^+ a^*$$

A questo punto non mi resta che considerare i casi che contengono 0 ripetizioni di $ab$:

- le prime due stringhe d'esempio ricadono in tali casi:

$$s_1 = a^6, s_2 = b^2 a^4$$

- non c'è modo di generarle partendo da $R_4$

# Espressione regolare III

- generalizzo $s_1, s_2$:

$$R_5 = a^+ \, R_6 = b^+ a^*$$

- considero altre stringhe degeneri, come quella formata da sole $b$:

$$R_7 = b^+$$

  ma mi accorgo che è generata da $R_6$

Alla fine non resta che assemblare le espressioni regolari:

$$R = R_4 \cup R_5 \cup R_6$$

# Chiusura rispetto al concatenamento

Prima di tutto verifichiamo se $L(R) \neq L(R^*)$:

- mi basta trovare un caso tale per cui la chiusura non sia valida

Osservo che posso ricavare la stinga nulla da $R^*$:

$$R^* \Rightarrow R^0 = \epsilon$$

Ma $\epsilon \notin L(R)$, quindi:

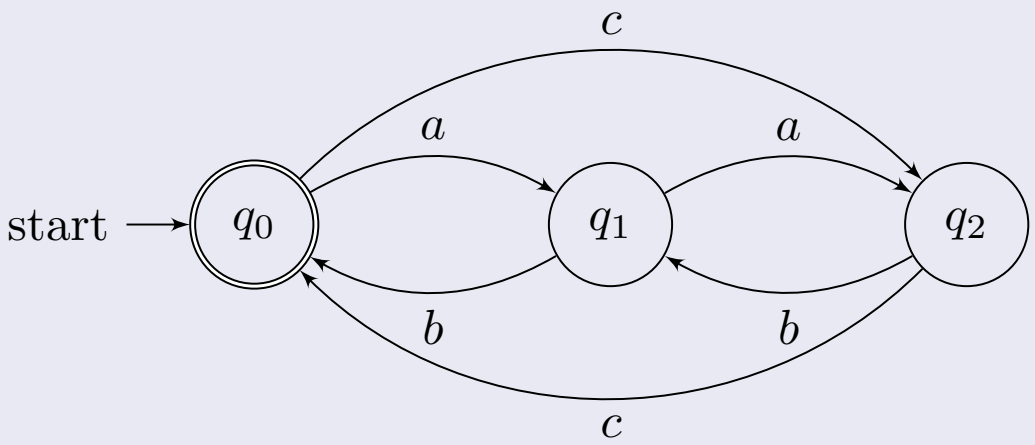
$$L(R) \neq L(R^*)$$

Se non trovassi tale caso devo dimostrare che le due espressioni regolari generano lo stesso linguaggio.

# Trasformazioni su automi I

Dato l'automa $A_1$, riconoscitore del linguaggio $L_1 \subset \Sigma^*_{a,b,c}$:

## Automa $A_1$




Data la proiezione $\pi$:

$$\pi(x) = \begin{cases} a & \text{for} \quad x = a \\ b & \text{for} \quad x = b \\ \epsilon & \text{for} \quad x = c \end{cases}$$

# Trasformazioni su automi II

Sia $A_2$ l'automa riconoscitore del linguaggio $L_2 \subset \Sigma^*_{a,b}$, ottenuto applicando la proiezione $\pi$ al linguaggio $L1$.
Si richiede di:

1. costruire l'automa deterministico minimo
2. calcolare l'espressione regolare $R_2$ generatrice del linguaggio $L_2$ partendo dall'automa precedentemente calcolato ed applicando l'algoritmo di Brzozowski e McCluskey

# Riconoscitore deterministico minimo di $L_2$

Nel testo ci sono due parole chiave, che richiedono delle proprietà fondamentali dell'automa che costruite:

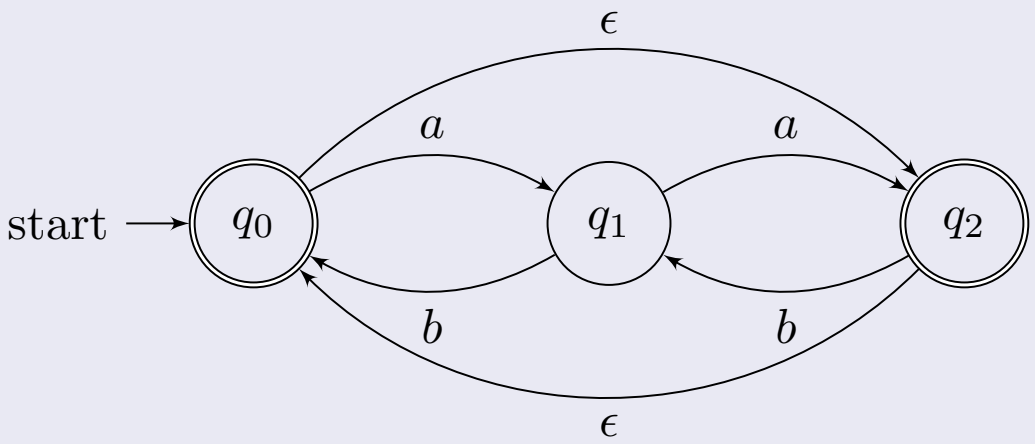
**deterministico** non esiste ambiguità sulla scelta del successore di un stato

**minimo** non esiste un altro automa riconoscitore di $L_2$ con un numero di minore di stati

# Proiezione

Applicando la proiezione otteniamo l'automa $TA_1$:

## Automa $TA_1$

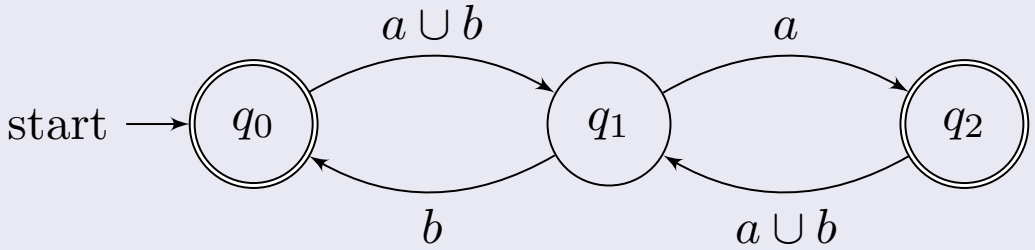



La presenza delle mosse spontanee rende l'automa $TA_1$ non-deterministico.

# Eliminazione mosse spontanee

Le mosse spontanee si eliminano facendone la chiusura transitiva; si ottiene l'automa $TA_2$:

## Automa $TA_2$




Esso è deterministico, ma non minimo:

- $q_0$ e $q_2$ sono equivalenti

# Minimizzazione automa I

Se non vedete che $q_0$ e $q_2$ sono equivalenti, si applica l'algoritmo di minimizzazione:

## Tabella degli stati

| | $a$ | $b$ |
|---|---|---|
| $q_0$ | $q_1$ | $q_1$ |
| $q_1$ | $q_2$ | $q_0$ |
| $q_2$ | $q_1$ | $q_1$ |

## Tabella di equivalenza

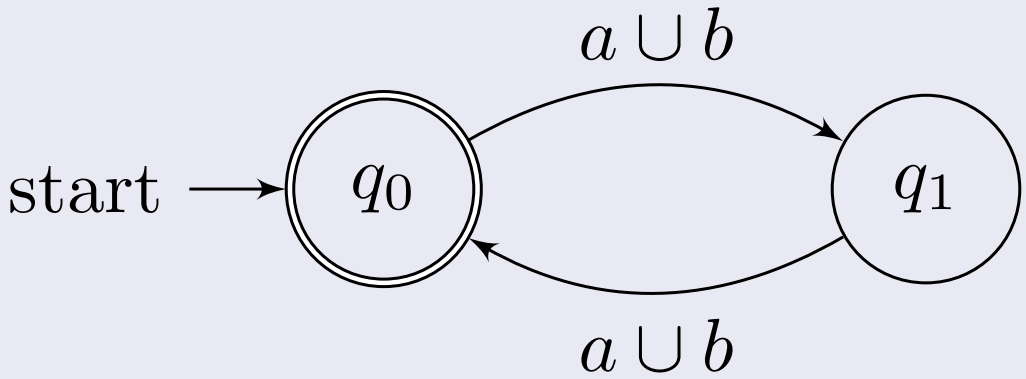
| | | |
|---|---|---|
| $q_1$ | $\not\equiv$ | $-$ |
| $q_2$ | $\equiv$ | $\not\equiv$ |
| | $q_0$ | $q_1$ |

L'algoritmo termina dopo un solo passo.

# Minimizzazione automa II

L'automa risultante minimo $A_2$ è:

## Automa $A_2$




# Espressione regolare I



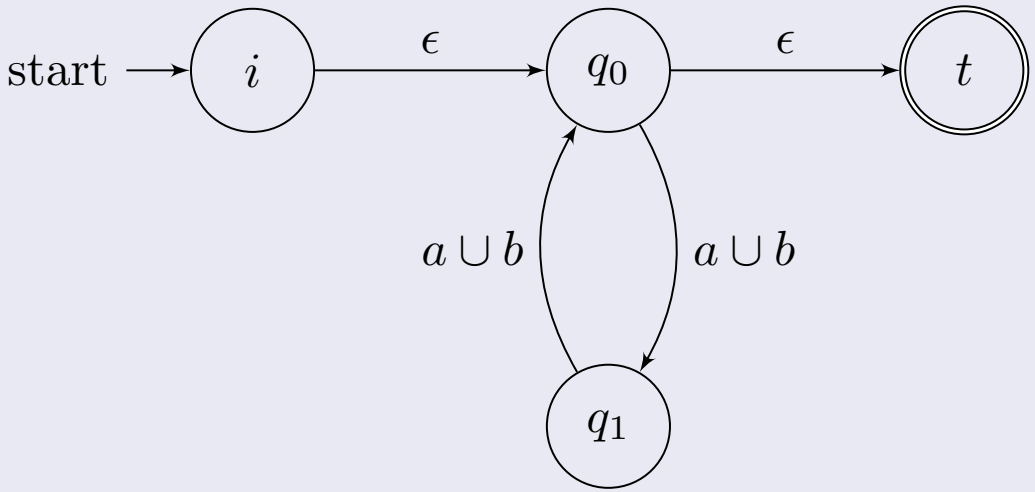
Il testo richiede esplicitamente di applicare un algoritmo noto:

- non possiamo calcolare "ad occhio" $R_2$
- dobbiamo applicare l'algoritmo di Brzozowski e McCluskey

# Espressione regolare II

Riscriviamo $A_2$ in modo che abbia un solo stato iniziale ed un solo stato finale:

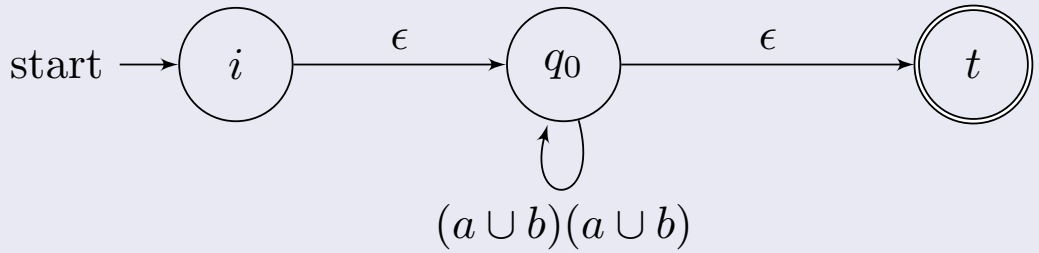
## Automa $AR_1$



# Espressione regolare III

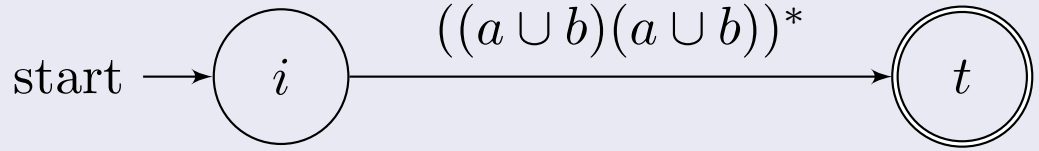
Fondiamo i nodi $q_0$ e $q_1$, ottenendo l'automa $AR_2$:

## Automa $AR_2$




Eliminiamo il loop:

## Automa $AR_3$




L'espressione regolare è indicata sull'arco $i \longrightarrow t$:

$$R_2 = ((a \cup b)(a \cup b))^*$$


# Sommario

# Grammatica delle espressioni I

Si consideri il linguaggio delle espressioni aritmetiche $L$ contenente:

- l'operatore infisso somma '+'
- l'operatore prefisso moltiplicazione '*mul*'
- gli operandi cifre '0', ..., '9'
- le parentesi tonde '(', ')'

Inoltre:

- la somma è associativa a sinistra:

$$a + b + c = (a + b) + c$$

- la moltiplicazione ha precedenza sulla somma:

$$a + * \; a \; b = a + (* \; a \; b)$$


# Grammatica delle espressioni II



Due possibili stringhe del linguaggio $L$ sono:

$$s_1 = 2 + 5 * 4 * 6\ 7$$
$$s_2 = *\ (3 + 2 +\ *\ 4\ 5)\ 8$$

Si richiede di:

1. progettare la grammatica in forma BNF
2. disegnare l'albero sintattico di $s_1$


# Progetto I



Prima di tutto consideriamo l'operatore con la precedenza minore:

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

Notiamo che la regola è ricorsiva a destra e a sinistra:

- la grammatica risulta ambigua
- non abbiamo forzato l'associatività dell'operatore $+$

La definizione corretta è:

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


# Progetto II



Ora consideriamo il non-terminale $T$; esso è legato all'operatore $*$:

$$T \longrightarrow * \; T \; T \mid F$$

Il non-terminale $F$ ci permette di iniziare una nuova espressione o di generare le cifre:

$$F \longrightarrow (E) \mid 1 \mid \ldots \mid 9$$

# Progetto III

Si noti come la grammatica dell'esercizio, $L$, e quella con gli operatori infissi associativi a sinistra, $M$, siano molto simili.

## Grammatica di $L$

$$E \longrightarrow E + T \mid T$$
$$T \longrightarrow * \; T \; T \mid F$$
$$F \longrightarrow (E) \mid 1 \mid \ldots \mid 9$$

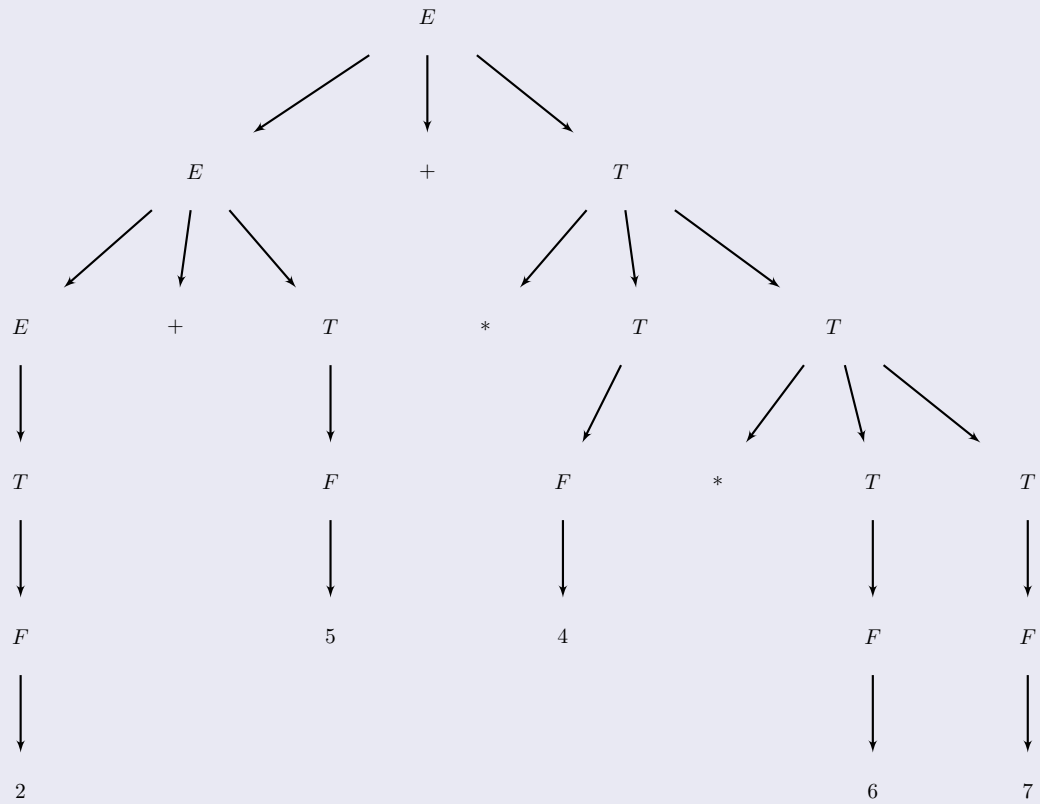
## Grammatica di $M$

$$E \longrightarrow E + T \mid T$$
$$T \longrightarrow T * F \mid F$$
$$F \longrightarrow (E) \mid 1 \mid \ldots \mid 9$$

# Alberi sintattici

Partiamo dalla radice della grammatica e generiamo tutti i non terminali che ci servono:

## Albero sintattico di $s_1$



# Ambiguità

Data la grammatica $G_0$:

## Grammatica $G_0$

$$S \to XY$$
$$X \to aXb \mid aX \mid ab$$
$$Y \to bYa \mid bY \mid ba$$

Si richiede di:

1. mostrare che $G_0$ è ambigua
2. costruire una grammatica equivalente non ambigua
3. disegnare l'automa riconoscitore di $L(G_0)$

# Ricerca ambiguità I

Analizziamo per primo il non-terminale $X$:

- il suo scopo è generare un linguaggio *quasi ben-parentitizzato*:

$$L(X) = \{a^n b^m \mid n \geq m \geq 1\}$$

- il progettista ha pero introdotto un'ambiguità:

$$X \Rightarrow aX \Rightarrow aaXb \Rightarrow aaabb$$
$$X \Rightarrow aXb \Rightarrow aaXb \Rightarrow aaabb$$

Passiamo ora al non-terminale $Y$:

- ha la stessa struttura del non-terminale $X$, cambiano solo i simboli terminali; deve essere per forza ambiguo:

$$Y \Rightarrow bY \Rightarrow bbYa \Rightarrow bbbaa$$
$$Y \Rightarrow bYa \Rightarrow bbYa \Rightarrow bbbaa$$

# Ricerca ambiguità II

Non resta che analizzare $S$:

- esso genera il linguaggio $L(S) = L(X) \,.\, L(Y)$
- $L(X)$ e $L(Y)$ hanno lo stesso alfabeto
- provo a cercare una stringa ambigua, generata in parte da $X$ ed in parte da $Y$:

$$S \Rightarrow XY \Rightarrow aXbY \Rightarrow aabbba$$
$$S \Rightarrow XY \Rightarrow aXbba \Rightarrow aabbba$$

# Rimozione ambiguità I

Conviene ragionare sulla struttura del linguaggio $L(G_0)$:

$$L(G_0) = \{a^n b^m b^q a^r \mid n \geq m \geq 1 \cap q \geq r \geq 1\}$$

Raccogliendo un po di termini il linguaggio è più chiaro:

$$\begin{aligned} L(G_0) &= \{a^n b^s a^r \mid n \geq 1 \cap s > r \geq 1\} \\ &= \{a^+ b^+ b^r a^r \mid r \geq 1\} \end{aligned}$$

# Rimozione ambiguità II

La grammatica $G_1$ per il linguaggio $L(G_0)$ è composta da due parti:

- un prefisso:

$$P \rightarrow AB$$
$$A \rightarrow aA \mid a$$
$$B \rightarrow bB \mid b$$

- un nido:

$$N \rightarrow bNa \mid ba$$

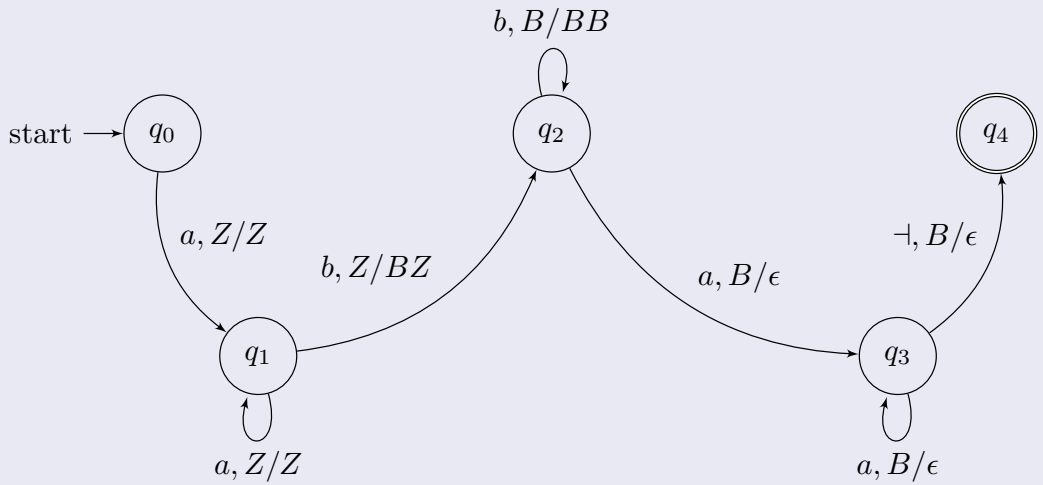
Concatenanti tra di loro:

$$S \rightarrow PN$$

# Automa riconoscitore

Siccome nel linguaggio c'è una struttura a nido, serve per forza un automa a pila:

## Automa $SA_1$



# Grammatiche $LL(k)$ e $LR(k)$

Data la grammatica $G_1$:

**Grammatica $G_1$**

$$S \rightarrow aSbS \mid aS \mid \epsilon$$

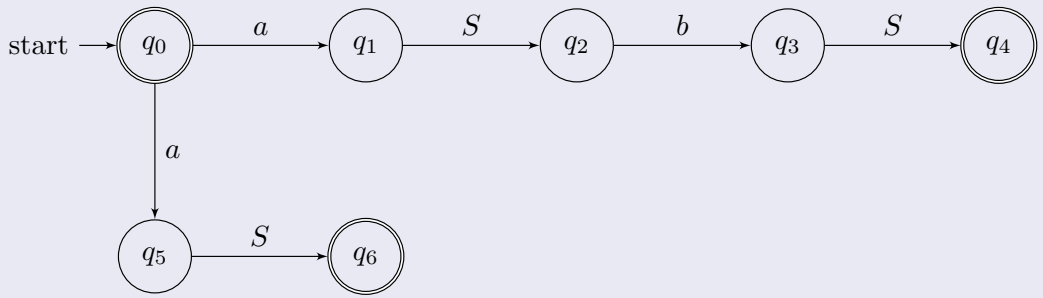
Si richiede di:

1. calcolare gli insiemi guida
2. verificare se la grammatica è $LL(1)$
3. costruire il riconoscitore dei prefissi ascendenti
4. verificare se la grammatica è $LR(1)$

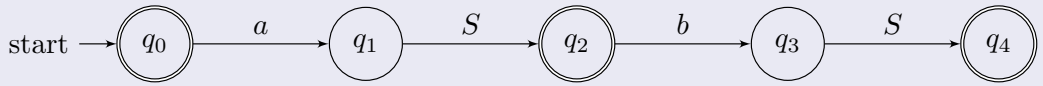
# Insiemi guida I

Per prima cosa scriviamo l'automa associato all'unica regola della grammatica:

## Automa $S$




## Automa $S$ determinizzato




# Insiemi guida II



Per calcolare gli insiemi guida servono:

- inizi dei non-terminali
- seguiti dei non-terminali

Nel nostro caso abbiamo un solo non-terminale, $S$:

## Inizi e seguiti

| Non-terminale | Inizi | Seguiti |
|---|---|---|
| $S$ | $\{a\}$ | $\{\dashv, b\}$ |

# Insiemi guida III

Per quanto riguarda gli insiemi guida, ha senso calcolarli solo per gli archi uscenti da stati con più successori:

- se lo stato $q_i$ ha un solo successore non c'è ambiguità sul cammino da seguire

Consideriamo quindi gli archi uscenti da $q_0$ e $q_2$:

## Insiemi guida

| Arco | Guida |
|---|---|
| $q_0 \longrightarrow q_1$ | $\{a\}$ |
| $q_0 \longrightarrow$ | $\{\dashv, b\}$ |
| $q_2 \longrightarrow q_3$ | $\{b\}$ |
| $q_2 \longrightarrow$ | $\{\dashv, b\}$ |


# Verifica $LL(1)$



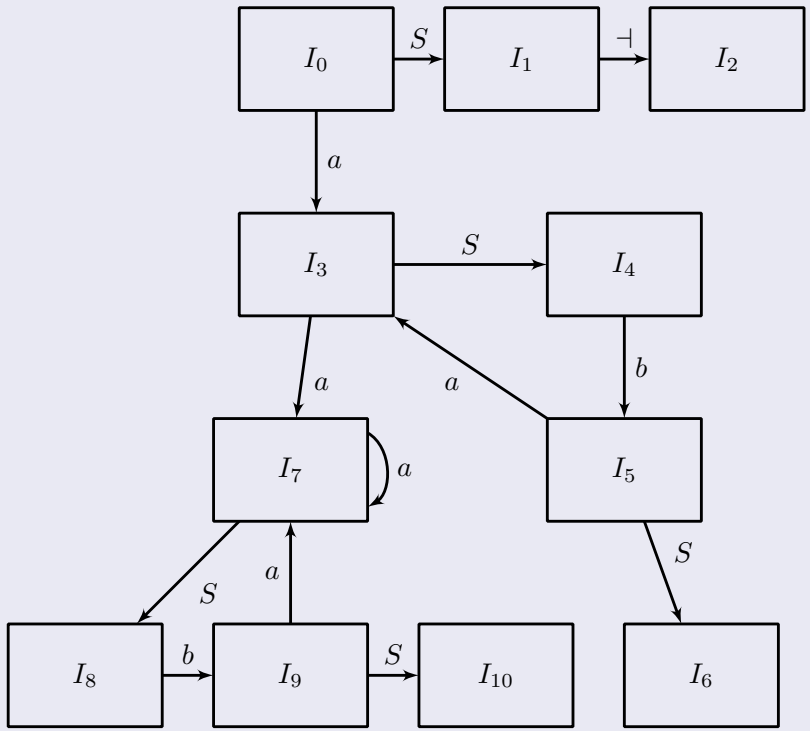
Applicando l'algoritmo devo verificare che gli insiemi guida degli archi uscenti da stati con più successori non abbiano elementi in comune:

- gli archi $q_2 \longrightarrow q_3$ e $q_2 \longrightarrow$ contengono entrambi il simbolo $b$
- la grammatica non è $LL(1)$

# Riconoscitore dei prefissi ascendenti

Esercizio molto meccanico, non c'è molto da dire:

## Automa $P_0$



# Verifica $LR(1)$

La grammatica $G_1$ non è $LR(1)$; infatti nello stato $I_8$:

- c'è un arco etichettato con $b$
- l'insieme di prospezione della candidata di riduzione $S \longrightarrow aS$ contiene $b$

In pratica, l'automa:

- non sa se procedere lungo il riconoscimento della regola $S \longrightarrow aSbS$
- o se riconoscere la regola $S \longrightarrow aS$


# Sommario

# Packing di numeri

Dato un numero decimale, si vuole eliminarne gli zeri non significativi:

## Esempio

$00203.0300 \longrightarrow 203.03$

$000.00 \longrightarrow 0.0$

Sapendo che il punto decimale non può essere omesso, si richiede di:

1. definire la trasformazione mediante uno schema di traduzione, senza utilizzare alcun attributo semantico
2. disegnare gli alberi sorgente e pozzo per il primo esempio
3. progettare un FSA trasduttore e verificare se è deterministico
4. progettare un SA trasduttore e verificare se è deterministico

# Schema di traduzione I

Il primo passo è identificare la struttura delle stringhe in ingresso:

- una parte intera ed una decimale:

$$S \rightarrow P.F$$

- più un prefisso sulla parte intera (gli zeri):

$$S \rightarrow ZP.F$$
$$Z \rightarrow 0Z \mid \epsilon$$

# Schema di traduzione II

La parte intera, a sua volta:

- è un numero:

$$P \rightarrow (0 \mid \ldots \mid 9)P$$

- ma in questo modo, generiamo anche il prefisso di zeri:

$$P \Rightarrow 0P \Rightarrow 03P \Rightarrow 03$$

- per impedirlo:

$$P \rightarrow (1 \mid \ldots \mid 9)Q \mid 0$$
$$Q \rightarrow (0 \mid \ldots 9)Q \mid \epsilon$$

# Schema di traduzione III

Per la parte frazionaria:

- può essere nulla (almeno uno zero):

$$F \rightarrow 0Z$$

- oppure essere un numero qualsiasi, terminato da zeri:

$$F \rightarrow Q(1 \mid \ldots \mid 9)Z$$

# Schema di traduzione IV

Il secondo passo è analizzare la grammatica sorgente, e modificarla in modo da cancellare le derivazioni non permesse dalla grammatica pozzo:

## Grammatica sorgente $G_0$

$$S \to ZP.F$$
$$Z \to 0Z \mid \epsilon$$
$$P \to (1 \mid \ldots \mid 9)Q \mid 0$$
$$Q \to (0 \mid \ldots \mid 9)Q \mid \epsilon$$
$$F \to 0Z \mid Q(1 \mid \ldots \mid 9)Z$$

## Grammatica pozzo $G_1$

$$S \to ZP.F$$
$$Z \to Z \mid \epsilon$$
$$P \to (1 \mid \ldots \mid 9)Q \mid 0$$
$$Q \to (0 \mid \ldots \mid 9)Q \mid \epsilon$$
$$F \to 0Z \mid Q(1 \mid \ldots \mid 9)Z$$

# Schema di traduzione V

La grammatica di traduzione $G_{0\rightarrow 1}$ è una rappresentazione alternativa della trasformazione:

## Grammatica di traduzione $G_{1\rightarrow 2}$

$$S \rightarrow Z P \dot{-} F$$

$$Z \rightarrow \frac{0}{\epsilon} Z \dot{} \mid \frac{\epsilon}{\epsilon}$$

$$P \rightarrow \left( \frac{1}{1} \mid \ldots \mid \frac{9}{9} \right) Q \mid \frac{0}{0}$$

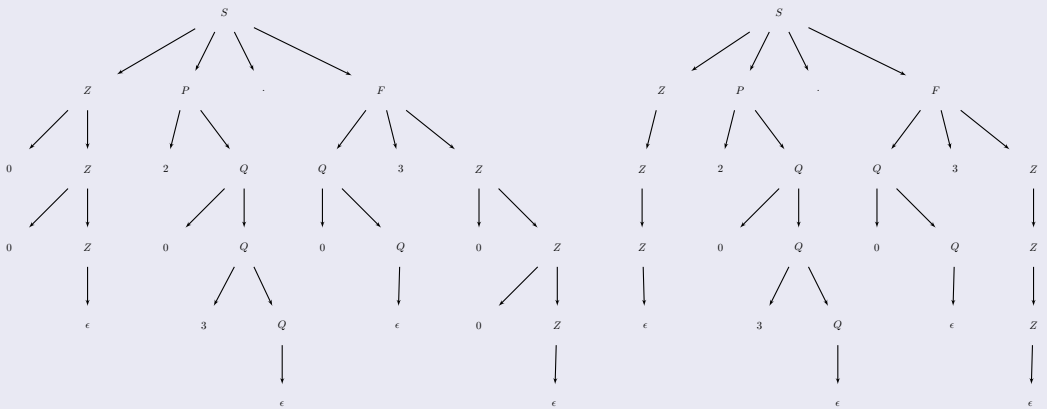
$$Q \rightarrow \left( \frac{0}{0} \mid \ldots \mid \frac{9}{9} \right) Q \mid \frac{\epsilon}{\epsilon}$$

$$F \rightarrow \frac{0}{0} Z \mid Q \left( \frac{1}{1} \mid \ldots \mid \frac{9}{9} \right) Z$$

# Alberi sintattici

I due alberi si costruiscono derivando "parallelamente" le regole corrispondenti nelle grammatiche $G_0$ e $G_1$:

## Alberi sorgente e pozzo di 00203.0300



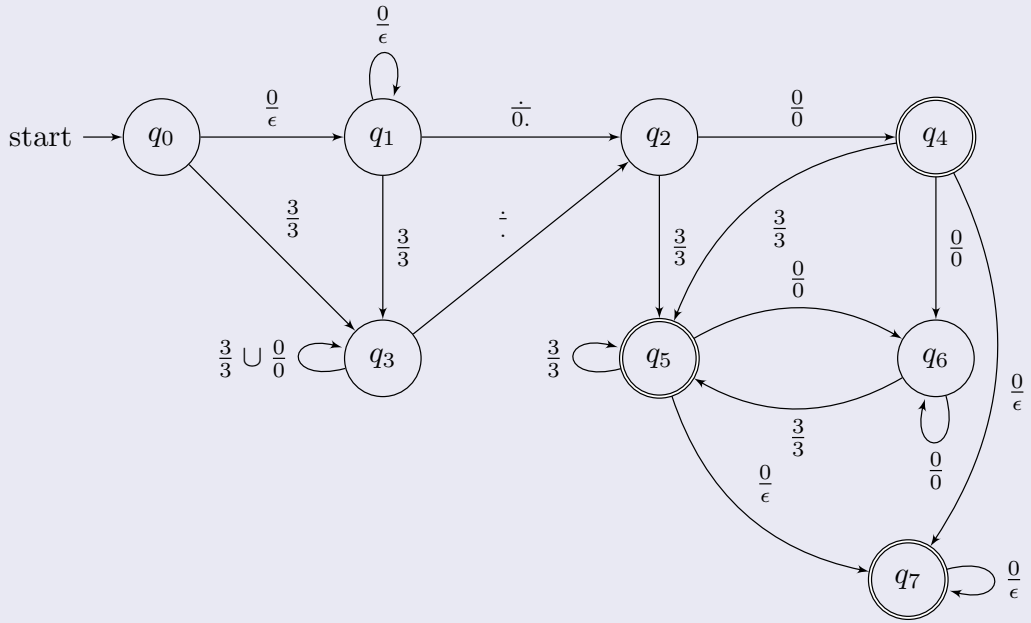
# FSA trasduttore I



Utilizzando un FSA, esso non può che essere indeterministico:

- deve speculare sul fatto che lo zero corrente sia parte del suffisso

## Automa $N_1$




# FSA trasduttore II



Note:

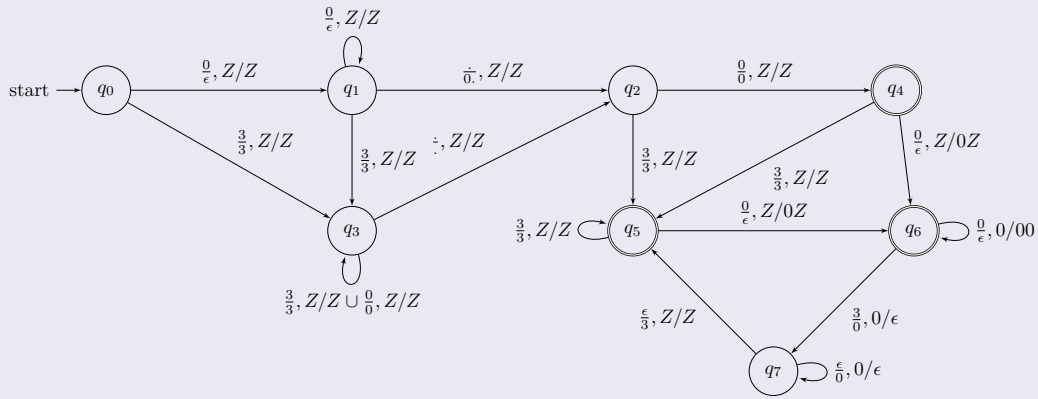
- il terminale 3 rappresenta una qualsiasi cifra diversa da 0

# SA trasduttore I

Un SA può utilizzare la pila per rimuovere la componente indeterministica:

- salva nella pila gli zeri e li stampa una volta risolta l'ambiguità

## Automa $N_2$




# SA trasduttore II



Note:

- il terminale 3 rappresenta una qualsiasi cifra diversa da 0
- lo stato $q_7$ rappresenta una classe di stati; esso serve per stampare gli 0 della pila e la cifra non nulla appena letta, serve quindi uno stato $q_{7_i}$ per ogni cifra non nulla

# Operazioni su insiemi I

Sia $G_0$ la grammatica:

**Grammatica $G_0$**

$$S \rightarrow \{I\}S \mid \{I\}$$
$$I \rightarrow e, I \mid e$$

Essa genera una lista di insiemi:

**Esempio**

$$\{e_1, e_5\}\{e_2, e_5\}\{e_1, e_5, e_6\}$$

I terminali $e_i$ rappresentano gli elementi degli insiemi e possiedono un attributo lessicale, *id*, che li identifica.


# Operazioni su insiemi II



Si richiede di:

1. progettare una grammatica ad attributi per calcolare l'intersezione degli insiemi
2. estendere la grammatica ad attributi per poter calcolare la differenza insiemistica di ogni insieme rispetto alla loro intersezione
3. disegnare i grafi delle dipendenze funzionali e stabilire quali tecniche di valutazione degli attributi possono essere applicate
4. scrivere, tramite pseudo-codice, il programma di un valutatore semantico

# Intersezione I



Per prima cosa riscriviamo la grammatica in modo che l'assioma non sia ricorsivo:

- in questo modo l'assioma contiene gli attributi "finali"

**Grammatica $G_1$**

$$S'_0 \longrightarrow S_1$$
$$S_0 \longrightarrow \{I_1\} S_2$$
$$S_0 \longrightarrow \{I_1\}$$
$$I_0 \longrightarrow e_1, I_2$$
$$I_0 \longrightarrow e_1$$

# Intersezione II

L'intersezione si calcola partendo dagli insiemi:

$s_i$ rappresentazione di un insieme, sintetizzato

Noto che:

- anche l'intersezione è un insieme
- la si può calcolare progressivamente

Si può quindi utilizzare lo stesso attributo, $s_i$, per rappresentare un'intersezione:

- parziale, se è associato ai non-terminali $S_i$
- totale, se è associato all'assioma

# Intersezione III

La grammatica $G_2$ decora la grammatica $G_1$ con i necessari attributi:

**Grammatica $G_2$**

| | |
|---|---|
| $S'_0 \rightarrow S_1$ | $s_0 = s_1$ |
| $S_0 \rightarrow \{I_1\} S_2$ | $s_0 = s_1 \cap s_2$ |
| $S_0 \rightarrow \{I_1\}$ | $s_0 = s_1$ |
| $I_0 \rightarrow e_1, I_2$ | $s_0 = id_1 \cup s_2$ |
| $I_0 \rightarrow e_1$ | $s_0 = id_1$ |

# Differenza I

La differenza avrà bisogno di un nuovo attributo, $d_i$:

- per calcolarlo avrò bisogno dell'intersezione totale, attributo dell'assioma

Per propagare l'intersezione totale, introduco un nuovo attributo, $i_i$:

- viene calcolato nell'assioma e poi propagato ai figli

Inoltre:

- $d_i$ è calcolato partendo da $i_i$, quindi deve essere per forza ereditato

# Differenza II

La grammatica $G_3$ introduce le necessarie modifiche in $G_2$:

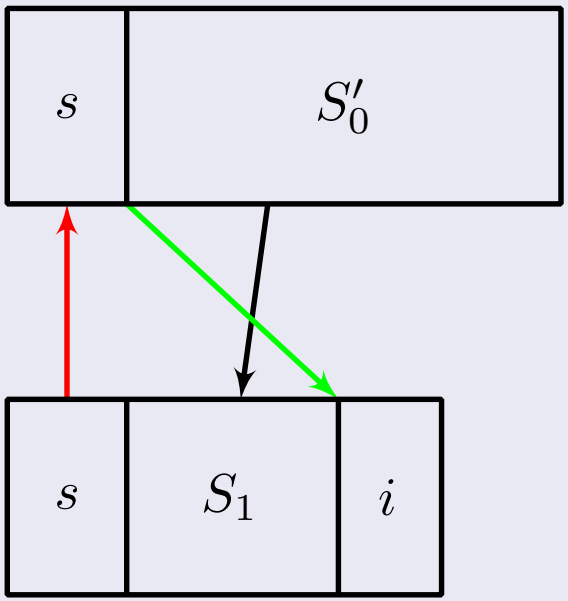
## Grammatica $G_3$

| | | |
|---|---|---|
| $S'_0 \to S_1$ | $s_0 = s_1$ | $i_1 = s_0$ |
| $S_0 \to \{I_1\} S_2$ | $s_0 = s_1 \cap s_2$ | $d_1 = s_1 \setminus i_0, i_2 = i_0$ |
| $S_0 \to \{I_1\}$ | $s_0 = s_1$ | $d_1 = s_1 \setminus i_0$ |
| $I_0 \to e_1, I_2$ | $s_0 = id_1 \cup s_2$ | |
| $I_0 \to e_1$ | $s_0 = id_1$ | |

# Dipendenze funzionali I

Si devono costruire 3 diagrammi, uno per ogni non-terminale della grammatica. Il primo è relativo all'assioma:

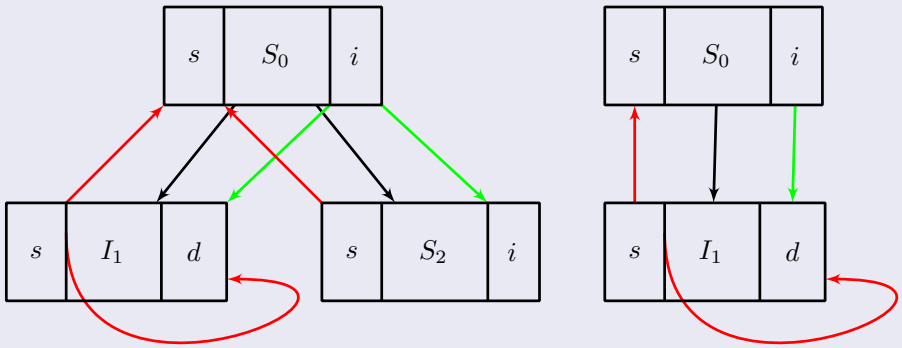
## Dipendenze di $S'_0 \longrightarrow S_1$



# Dipendenze funzionali II

Il secondo mostra le dipendenze delle regole $S_0 \longrightarrow \ldots$:

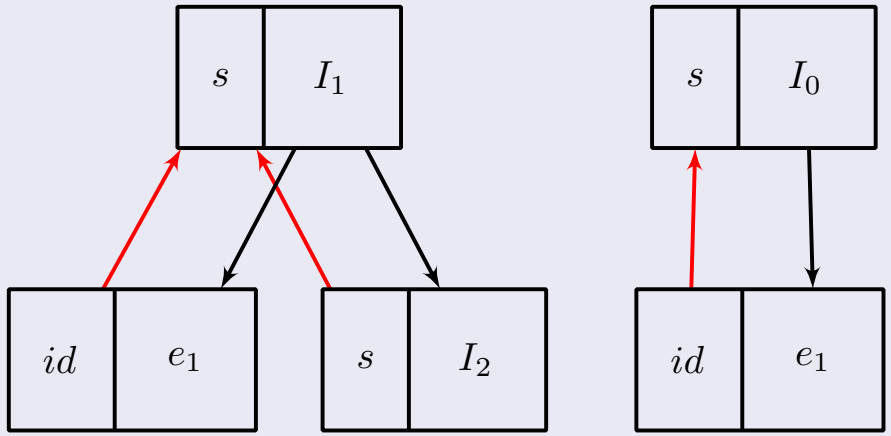
## Dipendenze di $S_0 \longrightarrow \ldots$



# Dipendenze funzionali III

Il terzo contiene i diagrammi delle dipendenze delle regole $I_0 \longrightarrow \ldots$:

## Dipendenze di $I_0 \longrightarrow \ldots$




# Dipendenze funzionali IV



Per quanto riguarda la tecnica di valutazione:

- l'intersezione deve essere calcolata per prima
- posso sfruttare un riconoscitore ascendente per costruirla passo-passo
- la differenza ha bisogno dell'intersezione totale
- devo calcolarla tramite una vista discendente, in una seconda passata

# Valutatore semantico



La grammatica $G_3$ genera un linguaggio regolare:

1. posso usare tool quali flex/bison per parsare l'input
2. si costruisce un AST, calcolando l'intersezione passo-passo
3. si visita l'AST in ordine, calcolando la differenza


# Sommario

# Stampa array I



Si vuole implementare una nuova istruzione nella macchina Acse in grado di stampare tutti gli elementi di un array:

## Esempio

```
int primes[3];

primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
write_array(primes);
```

## Output

```
2 3 5
```


# Stampa array II



Si richiede di:

1. definire i token (e le relative dichiarazioni in `Acse.lex` e `Acse.y`) necessari per ottenere le funzionalità richieste
2. definire le regole sintattiche (e/o le modifiche a quelle esistenti) necessarie per ottenere le funzionalità richieste
3. definire le azioni semantiche (e/o le modifiche a quelle esistenti) necessarie per ottenere le funzionalità richieste

# Token



Bisogna introdurre un solo nuovo token:

**Lexer**

```
"write_array" { return WRITE_ALL; }
```

E comunicare la sua presenza a bison:

**Parser**

```
%token WRITE_ALL
```

# Regole sintattiche



La nuova istruzione è un'operazione di I/O:

## Istruzioni di scrittura

```
read_write_statement :
      read_statement
    | write_statement
    | write_array_statement

write_array_statement :
  WRITE_ALL LPAR IDENTIFIER RPAR { ... }
```

# Azioni semantiche I



Prima di tutto bisogna sapere dove leggere:

## Recuperare l'array

```
write_array_statement:
  WRITE_ALL LPAR IDENTIFIER RPAR {
    t_axe_variable* array;
    int size;

    array = getVariable(program, $3);
    size = gen_load_immediate(
              program,
              array->arraySize);
```

# Azioni semantiche II

La seconda cosa da fare è sapere da dove iniziare a "contare":

## Inizializzazione contatore

```
int i;
t_axe_expression i_expr;
t_axe_label* head;

i = getNewRegister(program);
i_expr = create_expression(i, REGISTER);
gen_addi_instruction(
   program, i,
   REG_0, 0);

head = assignNewLabel(program);
```

# Azioni semantiche III



Il corpo del ciclo deve solamente stampare l'i-esimo elemento:

## Stampa elemento

```
int value;

value = loadArrayElement(
            program, $3,
            i_expr);
gen_write_instruction(program, value);
```

# Azioni semantiche IV

Infine verifichiamo se dobbiamo restare nel ciclo:

## Punto d'uscita

```
    int acc;

    gen_addi_instruction(program, i, i, 1);

    acc = getNewRegister(program);
    gen_sub_instruction(
      program, acc, i, size,
      CG_DIRECT_ALL);
    gen_bne_instruction(program, head, 0);
}
```


# Sommario


# Note finali



Ricordatevi che:

- bisogna passare tutte le parti del compito per passare l'esame
- la fretta è una cattiva consigliera

Domande? Chiarimenti?
(Ora o mai più)