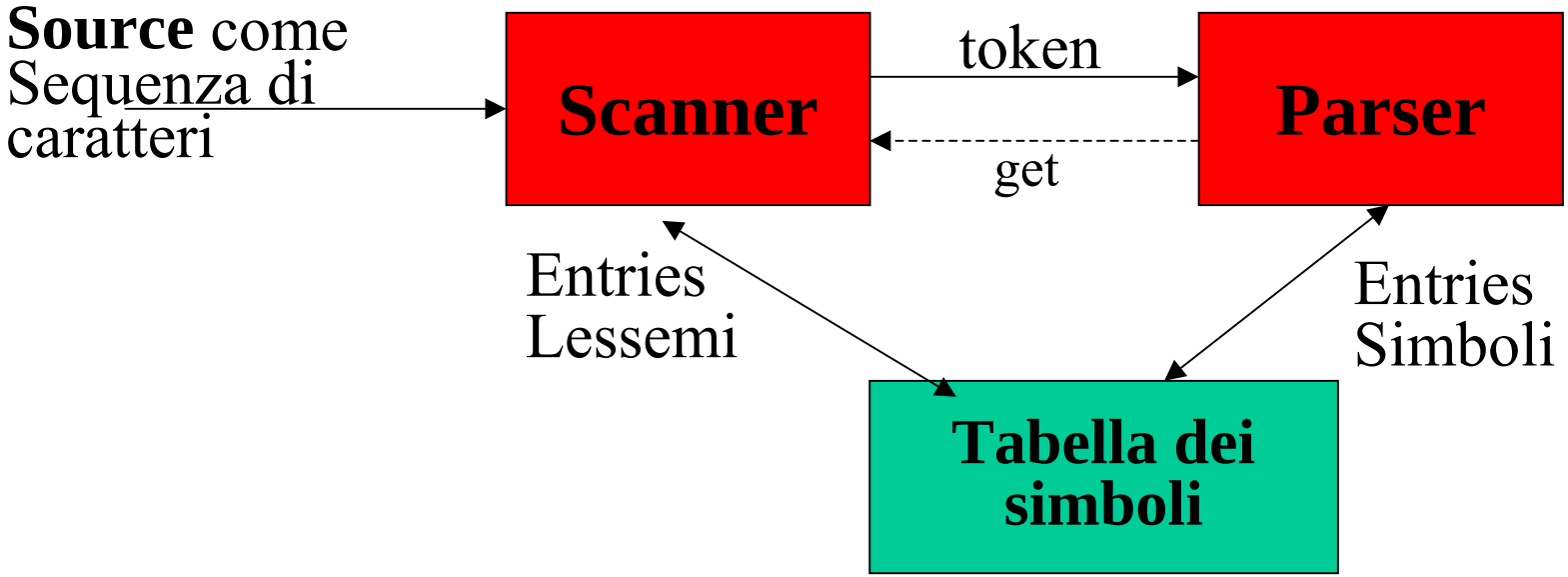
ANALISI LESSICALE

Controllare e riconoscere il linguaggio utilizzato per scrivere i simboli (identificatori, numerali, keywords, separatori, delimitatori,...)

linguaggio lessicale in generale semplice

linguaggio regolare,
grammatiche regolari o lineari
diagrammi di transizione

Una struttura una passata



Analisi lessicale (Scanner)
guidata dall'analisi sintattica (Parser)

token = categoria lessicale

lessema = valore di una categoria

pattern = regole lessicali con cui definire ogni categoria lessicale

esempio

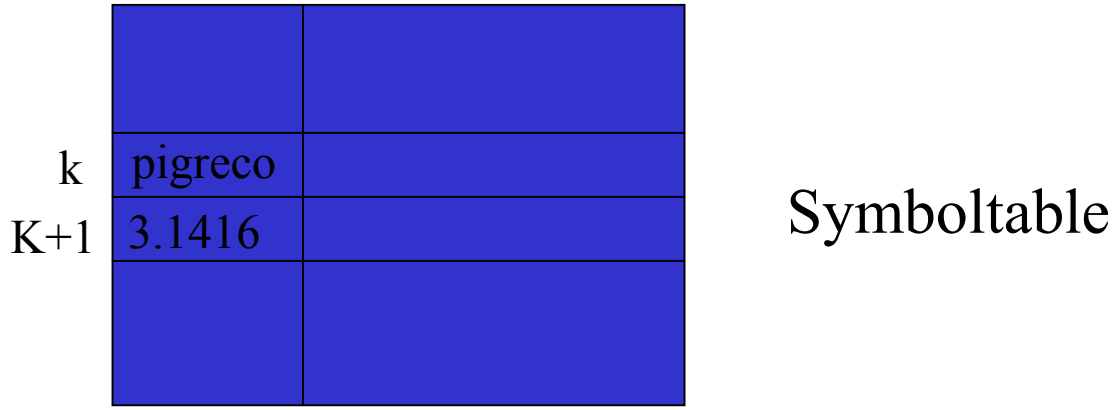
const pigreco = 3.1416

const id rel num

Dove finiscono i valori?

In effetti generiamo coppie: <**token**,attributo>

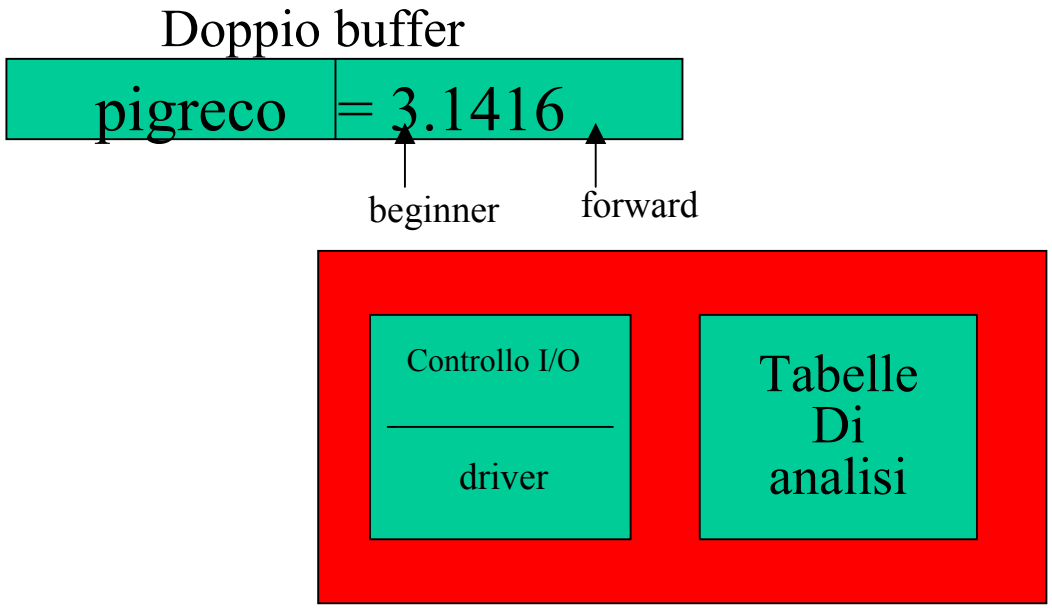
<**const**,> <**id**,k> <**rel**,=> <**num**,k+1>



I **pattern** sono definiti con vari formalismi:

- diagrammi di transizione
- grammatiche regolari

Possiamo costruire riconoscitori basati sui patterns



Espressioni e grammatiche regolari

Espressioni regolari E_Σ su Σ sono il

minimo insieme definito ricorsivamente da:

1. $\varepsilon \in E_\Sigma$
2. $a \in E_\Sigma, \ \forall a \in \Sigma$
3. $e_1.e_2 \in E_\Sigma, \ \forall e_1,e_2 \in E_\Sigma$
4. $e_1|e_2 \in E_\Sigma, \ \forall e_1,e_2 \in E_\Sigma$
5. $e^* \in E_\Sigma, \ \forall e \in E_\Sigma$
6. $e^i \in E_\Sigma, \ \forall e \in E_\Sigma$

Hanno significato L su 2^{Σ^*}

1. $[\varepsilon] = \{\lambda\}$
2. $[a] = \{a\}$
3. $[e_1.e_2] = \{uv \mid u \in [e1], v \in [e2]\} = [e1] \times. [e2]$
4. $[e_1|e_2] = \{u| \ u \in [e1] \cup [e2]\}$
5. $[e^*] = \{u| \ u \in \cup_{i \in N} [e]^i\}$
6. $[e^i] = \{u| \ u \in [e]^i\}$ *abbreviazione*

$\Sigma = \{a,b\}$

$\Sigma^* = \cup_{i \in N} \Sigma \times.\Sigma \ldots \times.\Sigma = \{\lambda,a,b,aa, ab, bb, ba, aaa, \ldots\}$

$2^{\Sigma^*} = \{u| \ u \subseteq \Sigma^*\} = \{\{\lambda\},\{a\},\{b\},\ldots,\{\lambda,a\},\ldots,\{\lambda,a,b,ab\},\ldots,\}$

0 | 1 | … | 9

[0 | 1 | … | 9] = {0,1,…,9}

(0 | 1 | … | 9)*

[(0 | 1 | … | 9)*] = {0,1,…,9, 00, 01,…, 3808, ...}

(1 | … | 9).(0 | 1 | … | 9)*

[(1 | … | 9).(0 | 1 | … | 9)*] = {1,…,9, 10, 11,…, 3808, ...}

Grammatica (regolare) **su Σ**

$G = \langle V, \Sigma, s \in V, P \rangle$

V *non terminali* (token ed ausiliari)

Σ *terminali*

s *simbolo distinto*

$P \subseteq V \times E_{\Sigma \cup V}$ *produzioni*

G e' una grammatica, affinche' sia
regolare dobbiamo aggiungere:

- V sono totalmente ordinati,<
- per ogni $v ::= e \in P$,
  $e \in E_{\Sigma \cup \{v' < v\}}$

Grammatiche come equazioni tra linguaggi

$$e \in \mathbf{E}_{\Sigma} \Longrightarrow \mathbf{L}(e) \in 2^{\Sigma^*}$$

$$v ::= e \Longrightarrow v = \mathbf{L}(e) \in (V \times 2^{\Sigma^*})$$

$$\{v_1 ::= e_1, \ldots, v_k ::= e_k\} \Longrightarrow \{\underline{\mathbf{L}}(v_1), \ldots, \underline{\mathbf{L}}(v_k) \mid \forall i, \underline{\mathbf{L}}(v_1) \in 2^{\Sigma^*} \textit{ and } (v_i \equiv \mathbf{L}(e_i))_{\{\underline{\mathbf{L}}(v_1)/v_1, \ldots, \underline{\mathbf{L}}(v_k)/v_k\}}\}$$

$$G = \langle V, \Sigma, s \in V, P \rangle \Longrightarrow \underline{\mathbf{L}}(G) = \underline{\mathbf{L}}(s)$$

Esempio

Relazione tra grammatiche, espressioni e diagrammi

Un lessico per i numerali con grammatiche regolari

```
simple::= digit digit*
fract::= simple.simple
exp::= fract E simple |
        fract E (+|-) simple
num::= simple | fract | exp
digit::= 0 | 1 | … | 9
```

```
s::= d d*
f::= s.s
e::= f E s |
      f E (+|-) s
n::= s | f | e
d::= 0 | 1 | … | 9
```

chi e' Σ ?
quali strutture sono in G ?

Omettiamo il simbolo "." dell'operatore di giustapposizione

G = <V,Σ,s∈ V,P>

V = {simple, fract, exp, num}
Σ = {0, 1, …, 9 , . ,+, -}
s = num

Notare l'uso di metasimboli:
(,) per comporre espressioni regolari

Riconoscere il lessico espresso da una regolare

Automa a stati finiti

$<S, \Sigma, move: S \times \Sigma' \dashrightarrow S', S_0 \subseteq S, F \subseteq S>$

per S finito

NFA (nondeterministico)

$\Sigma' = \Sigma \cup \{\varepsilon\}$ mentre $S' = 2^S$

S_0 ha cardinalita' > 1

DFA (deterministico)

$\Sigma' = \Sigma$ mentre $S' = S$

S_0 e' singoletto

La funzione *move*

Possiamo usare grafi simili ai diagrammi [G] o tabelle [T] per descrivere la funzione finita *move*

Ad ogni stato S corrisponde:

G un vertice del grafo

T un'entry della tabella con tante colonne quanta la cardinalita' di Σ (+1 per nondeter.) tante righe quanti gli stati

Ad ogni $\langle\langle s,a\rangle,S\rangle \in$ *move* corrisponde:

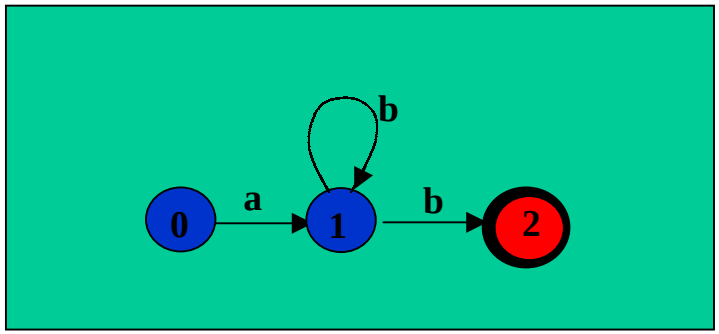
G un'arco $\langle s,t\rangle$ etichettato a per ogni $t \in S$

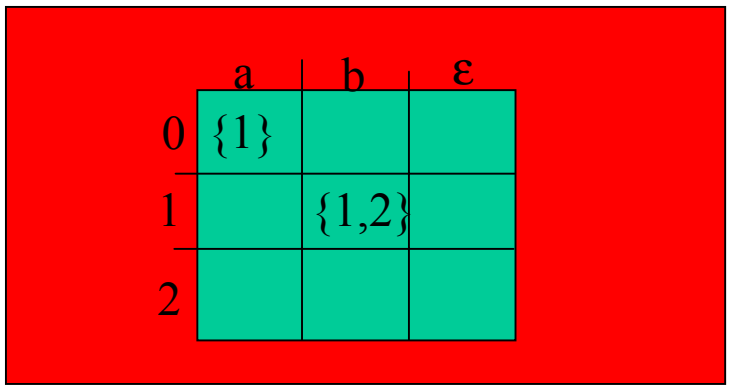
T il valore S per l'entry $\langle s,a\rangle$

Esempio: **consideriamo l'automa L**

$<S = \{0,1,2\},$
$\Sigma = \{a,b\},$
$move = \{<<0,a>\{1\}>, <<1,b>\{1,2\}>\}$
$s_0 = 0,$
$F = \{2\} >$

E' questo un automa deterministico
oppure automa nondeterministico ?


b
a
b
0
1
2


a
b
ε
0
{1}
1
{1,2}
2

DOVE POSSIAMO DARE SIGNIFICATO AD UN AUTOMA A STATI FINITI

$L = \langle S, \Sigma, move: S \times \Sigma' \dashrightarrow S', s_0 \in S, F \subseteq S \rangle$

sui linguaggi regolari $\Re$ su 2^{Σ^*}

Usiamo funzioni di decisione

sem(L) = $f \in \Sigma^*$ ---> {*accept*,*noaccept*}

tale che: per ogni $c_1c_2....c_n \in \Sigma^*$

$f(c_1c_2....c_n)$ = *accept*

se e solo se

$c_1c_2....c_n$ appartiene al linguaggio riconosciuto

definiamo la relazione $\Longrightarrow$

$\gamma \in \Sigma^*, s \in S \Longrightarrow \gamma', s'$

se e solo se:

$\gamma = c\gamma'$ and $s' \in move(c,s)$

oppure

$\gamma = \gamma'$ and $s' \in move(\varepsilon,s)$

Sia ==>* la chiusura transitiva di ==>

Allora:

$$f(\gamma) = \begin{cases} \textit{accept} \ \textbf{se} \ \gamma, s_0 ==>^* \lambda, s \in F \\ \textit{noaccept} \ \textbf{se per nessun} \ s \in F, \ \gamma, s_0 ==>^* \lambda, s \end{cases}$$

Da un'espressione regolare E a un NFA che riconosce il linguaggio L(E)

1. ε

$<\{s_0\}, \{\}, \{\}, s_0 \in S, \{s_0\}>$

2. a per ogni $a \in \Sigma$

$<\{s_0,s_1\}, \{a\}, \{<<s_0,a>,s_1>\}, s_0 \in S, \{s_1\}>$

3. e1.e2 per ogni e1,e2∈ E

e1: $<S_1, \Sigma_1, M_1, s_1, \{f_1\}>$
e2: $<S_2, \Sigma_2, M_2, s_2, \{f_2\}>$

e1.e2: $< S_1 \cup S_2, \Sigma_1 \cup \Sigma_2,$
$M_1 \cup M_2 \cup \{<< f_1, \varepsilon >, \{s_2\}>\},$
$s_1, \{f_2\}>$

4. e1|e2 per ogni e1,e2∈ E

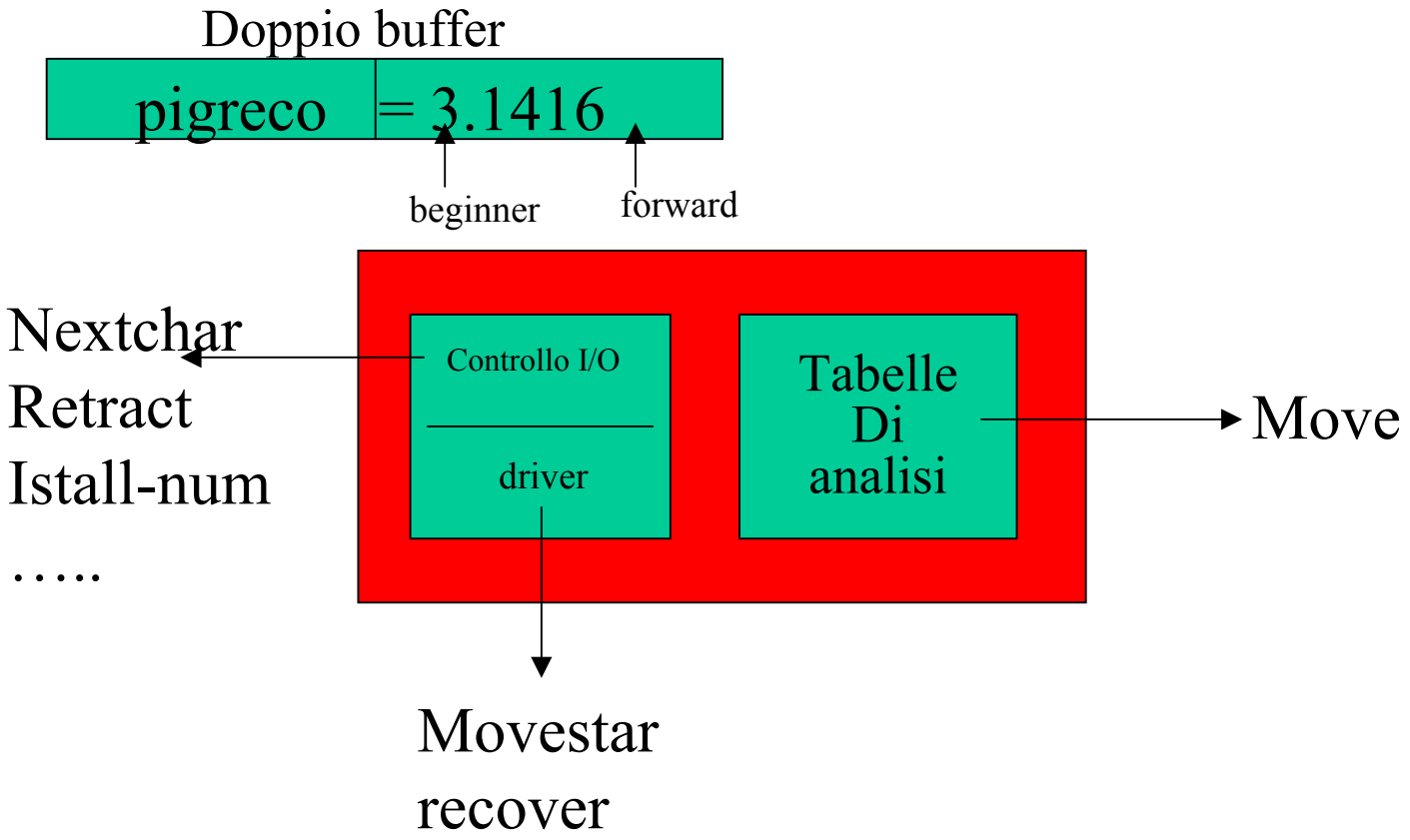
e1: $<S_1, \Sigma_1, M_1, s_1, \{f_1\}>$
e2: $<S_2, \Sigma_2, M_2, s_2, \{f_2\}>$

e1|e2: $< S_1 \cup S_2 \cup \{s_{new}, f_{new}\}, \Sigma_1 \cup \Sigma_2,$
$M_1 \cup M_2 \cup \{<< s_{new}, \varepsilon >, \{s_1, s_2\}>,$
$<< f_1, \varepsilon >, \{f_{new}\}>$
$<< f_2, \varepsilon >, \{f_{new}\}>\},$
$s_{new}, \{f_{new}\}>$

5. e*per ogni e∈ E

e: $<S, \Sigma, M, s, \{f\}>$

e^* : $< S \cup \{s_{new}, f_{new}\}, \Sigma,$
$M \cup \{<< s_{new}, \varepsilon>, \{s, f_{new}\}>,$
$<< f, \varepsilon>, \{s, f_{new}\}>\},$
$s_{new}, \{f_{new}\}>$


Doppio buffer
pigreco
= 3.1416
beginner
forward
Nextchar
Retract
Istall-num
…..
Controllo I/O
driver
Tabelle
Di
analisi
Move
Movestar
recover

Tre diffrenti driver:

1) **Diretto** Esecutore
   (gestisce il nondeterministismo con apposite strutture per il backtrack)
2) **Emulato** Interprete
   (interpreta il nodeterminismo come don’t care: usa stati che sono insiemi)
3) **Tradotto** Compilatore
   (compila il nodeterminismo: traduce un nondeterministico in un deterministico)

1) Un driver per NFA (DA)

```
Answer driver()
    {states=push([Clos({s0}),pointerInput()]);
     repeat
        answer='accept';
        [s,input]=pop(states);
        nextchar(c);
        while(c≠eof) and (s≠⊥) {
            S=move[s,c];
            push([Clos(S),pointerInput()]);
            [s,input]=pop(state);
            nextchar(c);}
        if (s∉ F) or (c≠eof) answer='noaccept';
     until  emptystack() or (answer='accept');
     return (answer);
    }
```

States: stack di controllo
Push:inserisce [S,&]
S: insieme di stati
&: punatore input
Pop: rimuove uno stato da S
e copia & nell'input
Se S e' singoletto l'intera coppia [S,&] e' rimossa dallo stack
⊥: valore calcolato da move[s,c] quando la tabella non ha entry per move[s,c]

2) Un simulatore per NFA

move* ci fornisce un riconoscitore

```
Answer movestar()
        {S=Clos({s0});
         nextchar(c);
         while(c≠eof) and (S≠∅) {
                  S=move1[S,c]);
                  nextchar(c)};
         if (S∩F) ≠ ∅  return 'accept';
         else if return 'noaccept';
        }
```

Lineare
(passo exp)

Esercizio 1

Eliminiamo il nondeterminismo della relazione

$$f(\gamma) = \begin{cases} \textit{accept} \ \textbf{se} \ \text{move}^*(\gamma) \cap F \neq \{\} \\ \textit{noaccept} \ \textbf{altrimenti} \end{cases}$$

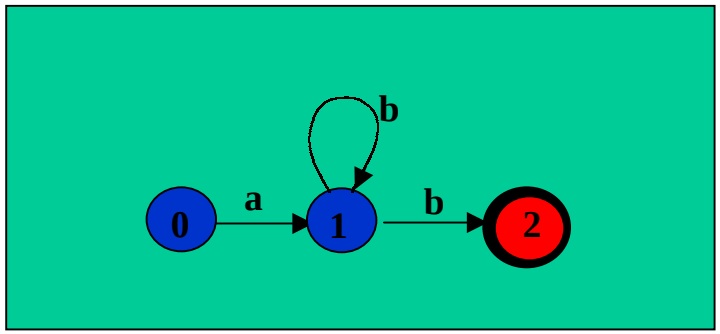
move

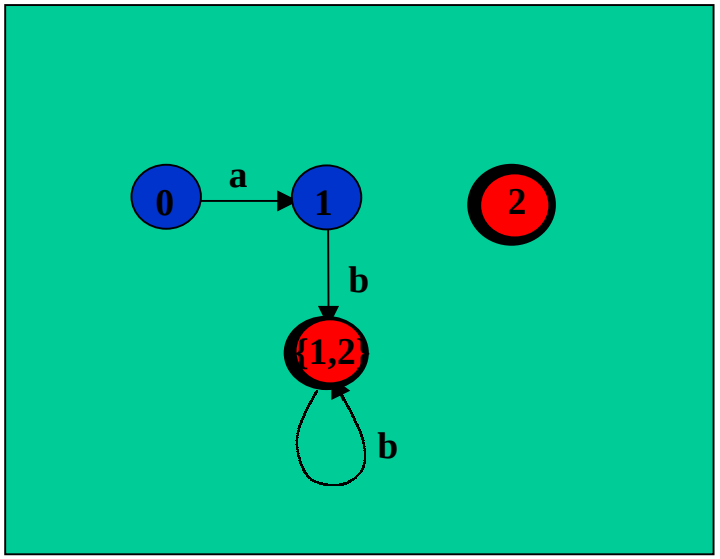
	a	b	ε
0	{1}		
1		{1,2}	
2			

move*

	a	b	ε
0	{1}		
1		{1,2}	
2			
{1,2}		{1,2}	

graficamente





Alcuni stati

possono diventare inutili

possono risultare ridondanti

La funzione move*:

$$move1(S,c) = Clos(\cup_{s \in Clos(S)} \{move(s,c)\})$$

$$Clos(S) = S \cup Clos(\cup_{s \in S} move(s,\varepsilon))$$

Allora:

$$move^*(c) = move1(\{s_0\},c)$$

$$move^*(c_1,\ldots,c_{n-1},c_n) = move1(move^*(c_1,\ldots,c_{n-1}),cn)$$

Star: Set -> (Σ -> Set)

Osservare: Star non contiene colonna ε

Star [S] c = S'

$Star[S]+ = \cup_{c \in \Sigma} Star[S]c$

merge (move,S) = merge-row({move[s] | s∈ S})

$\forall_{c \in \Sigma}$ merge-row({$R_1,\ldots,R_k$}) c = ($\cup_{1 \le i \le k} Clos(R_i\ c)$)

	digit	.	ε
0	1		1
1	3	2	{1,3}
2		2	
3	0	4	
4	4	5	
5	6		
6	6		

move

merge-row({0,1,3})= {0,1,3} | {2,4} |

3) Un convertitore NFA in DA

```
Table movestar()
        {EntryStar = ∅;
         List= Clos(S0);
         while (List≠emptylist) {
                 S=firstout(List);
                 if (S∉ EntryStar) {
                         add(S,EntryStar);
                         Star[S]= merge(move,S)};
                 List=List+Star[S]+;
          return Star;
        }
```

Exp
(passo lineare)

Esercizio 2

Una seconda tecnica per costruire riconoscitori

e’ basata sui

diagrammi di transizione

Diagrammi di transizione su Σ

sono grafi diretti etichettati $G(V,E,\Sigma)$

- vertici, detti stati, bipartiti in:
  - interni
  - finali
- archi $E \subseteq V \times V$ (source,target)
- per ogni $e \in E$, $label(e) \in \Sigma$

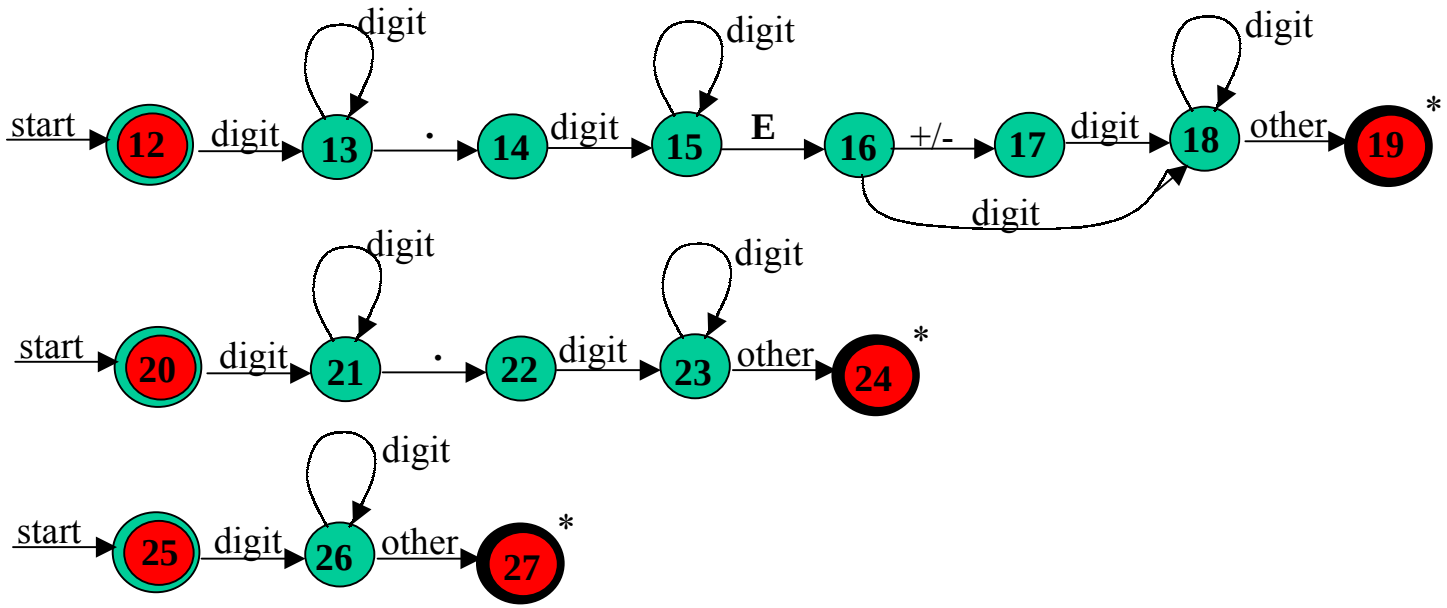
- nondeterministici

- deterministici

label(e1) = label(e2) only if
source(e1)≠source(e2)

Esempio (continua)

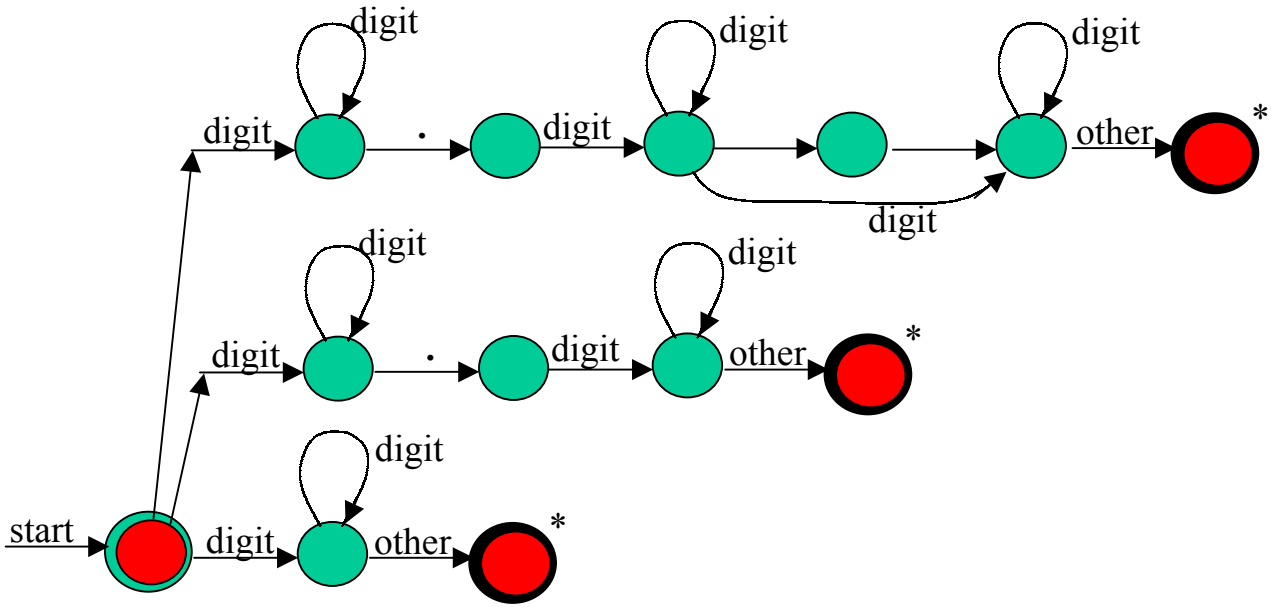
un lessico per i numerali con **diagrammi di trasizione**



I diagrammi sono deterministici??
Come mai tanti diagrammi??
Le annotazioni * e other fanno parte della definizione??

I diagrammi non sono deterministici

Infatti essi corrispondono al diagramma:



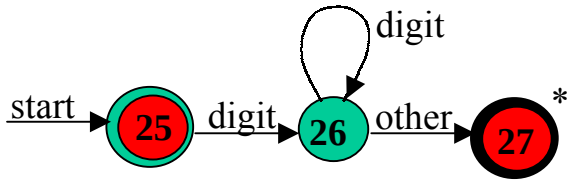
COSTRUIAMO I RICONOSCITORI

diagrammi sintattici sono gia' adatti

ogni transizione puo' essere descritta
da una struttura del tipo

if *predicate* **then** *action*

1. scriviamo **un frammento di codice** (nel metalinguaggio)
per ogni stato di ogni diagramma



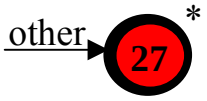
stati interni

```
25 : if (isdigit(c)) state=26;
     else if state=fail()

26 : if (isdigit(c)) state=26;
     else if state=27
```

C contiene l'ultimo carattere letto

stati finali




```
27 :retract(1);
     install-num();
     return(num)
```

Abbiamo usato operazioni

per il controllo dell'I/O

per l'update della Symboltable

La definizione della corretta forma per

predicate e action

dipende dal metalinguaggio:
strutture dati e di controllo permesse.

abbiamo usato C:
i condizionali hanno quella forma

mentre variabili e funzioni sono:

c: variabile globale contenente carattere del buffer su cui e'posizionato forward (ultimo carattere letto)

nextchar(): avanza forward di 1

retract(k): indietreggia forward di k posizioni

state: variabile contenente stato correntemente attraversato

Start: variabile contenente stato iniziale corrente riconoscimento

install-num(): aggiorna la symboltable

fail(): gestisce il fallimento

isdigit(a): riconosce un simbolo *digit*

2. **componiamo tutti questi frammenti**

useremo un *case_statement* come discriminante

useremo un iteratore per
leggere un carattere per volta
riesaminare caratteri dopo *fail()*

otteniamo la seguente struttura di programma

```
int state = 0;
int lexical-value;

token nexttoken()
{while(1) {
        switch (state) {
        case 0:....
        ..............
        case 25: if (isdigit(c)) state=26;
                else if state=fail()
        case 26: if (isdigit(c)) state=26;
                else if state=27
        case 27: retract(1);
                install-num();
                return(num)
        ..............
                }
        }
}
```

Osservazioni sull'ambiente di lavoro del programma

Il parser invoca nexttoken() attendendosi una coppia <token,valore>.
Il programma restituisce una coppia?

NO: restituisce solo il token (num,id...).
pero' una variabile *lexical-value* puo'
essere resa globale e condivisa dal parser

install-..: oltre a modificare la *sym-table* pone in
lexical-value l'entry relativa al nuovo token

Il **nonderminismo** sulla scelta dei vari diagrammi e'gestito:

ordinando i diagrammi in accordo allo stato iniziale

saltando da un diagramma all'altro in caso di fallimento [*fail()*]

```
int fail()
        {forward = beginner;
         switch (start) {
                case 0:....
                ..........
                case 12: start=20; break;
                case 20: start=25; break;
                case 25: recover(); break;
                        }
         return start;
         }
```

ri-iniziamo la scansione della sequenza di caratteri

Partiamo da un altro diagramma

Stack
beginner

Quando abbiamo esaminato tutte le alternative

Esponenziale

Qual’e’ la vera routine di gestione degli errori lessicali???

recover()

Differenze tra *fail()* e *recover()*

Proprieta' interessanti (1)

L1 e L2 sono regolari.

Unione: $L1 \cup L2$ e' regolare? SI

Prodotto: $L1 \times L2$ e' regolare? SI

Intersezione: $L1 \cap L2$ e' regolare? SI

Complemento: C(L1) e' regolare? SI

L e' un linguaggio

Decidibilita': L e' regolare? NO

Proprieta'interessanti (2)

L e' regolare
$A=<S,\Sigma,m,s_0,F>$ e' automa per L

Th.(di iterazione)
Se $x \in L$, $|x| \geq \#S$,
Allora: $\exists u,w,v$ tali che $x=u.w.v$, $u.w^k.v \in L$

Esercizi

fine