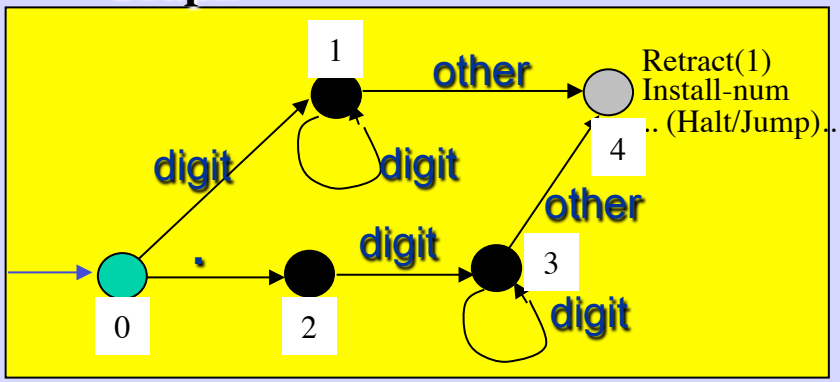
How to build Scanner for a Lexics L
Other Techniques

- *Transition Diagrams*
- *Minimization*
- *Dotted Automata*

Transition Diagrams on Σ − 1

- Are used in *by hand* constructions, of recognizers.
- Lexics must be *quite simple* to guarantee correctness by eye.
- Consist of directed, labeled, possibly, annotated, ***Graphs***,
- And in a ***Codification Code*** of the graph.

Graph



Codification Code

- For each *node* one *case statement* is generated

– case 0: **if** (c==digit) state = 1;
else if (c==dot) state = 2;
else fail(0);

– case 1: **if** (c==digit) state = 1;
else state = 4;

– case 2: **if** (c==digit) state = 1;
else fail(2);

– case 3: **if** (c==digit) state = 3;
else state = 4;

– case 4: { retract(1);
install-num();
…}

Transition Diagrams on Σ − 2

- Diagrams/Codes can be composed to construct a scanner

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

token nexttoken()
      {while(1) {
          switch (state) {
                case 0: …
                case 1: …
                case 2: …
                case 3: …}
                }
}
```

A list of auxiliary structures and functions that have to be implementated

forward: input pointer
c: character to be read
nextchar(): go to next character
retract(k): go k characters back
state: last traversed state
Start: initial state
install-num(): update symbol table
fail(state)-recover(state): management of the failure conditions
digit, dot, ... : predicates for special characters
I/O: interface controller
…

Dotted Automata: From Regular Expressions To DFA

- **Dotted Automata:How to obtain Deterministic Automata directly from Regular Expressions E_Σ**
- Let *e* be the input RE. It results an output DFA, A_e such that $L(e)=L(A_e)$
  - A_e **Initial State**: $I_0=C_s(\{_\Delta e\})$,
  - Definition of the ***Closure Function*** C_s (by induction on *e*):
    - $C_s(S)= \cup_{\Delta\beta \in S} C(_\Delta \beta)$
      - $C(_\Delta \varepsilon)= \{_\Delta\}$;
      - $C(_\Delta b\beta)= \{_\Delta b\beta\}$, $\forall\, b\in\Sigma,\ \beta\in E_\Sigma$;
      - $C(_\Delta e^*\beta)= C(_\Delta ee^*\beta)\cup C(_\Delta\beta)$;
      - $C(_\Delta(e1|e2)\beta)= C(_\Delta e1\ \beta)\cup C(_\Delta e2\ \beta)$;
      - *additional cases can be added for optional, additional, RE operators*
  - Definition of the A_e's *move, denoted* **M**, and of the remaining A_e states $\mathbf{I_j}$:
    - **M(Ii**,a)=**Ij sse**
      - $a\in first(\boldsymbol{Ii})=\{a \mid {}_\Delta a\beta \in \boldsymbol{Ii},\ a\in\Sigma\}$
      - **Ij** $= C_s(\{_\Delta\beta \mid {}_\Delta a\beta \in \boldsymbol{Ii},\ a\in\Sigma\})$
  - **Final State**: $\{\boldsymbol{Ij} \mid {}_\Delta\in \boldsymbol{Ij}\}$

- Induction is on the concatenation stucture of *e*
- $_\Delta$ is the dot symbol that is used to mark an item
- $_\Delta e$ is called dotted item
- $_\Delta$ is the empty item

Apply it to expression

$e = ab^*b$

Compare the technique with the Thompson's one

Dotted Automata: From Regular Grammars To DFA

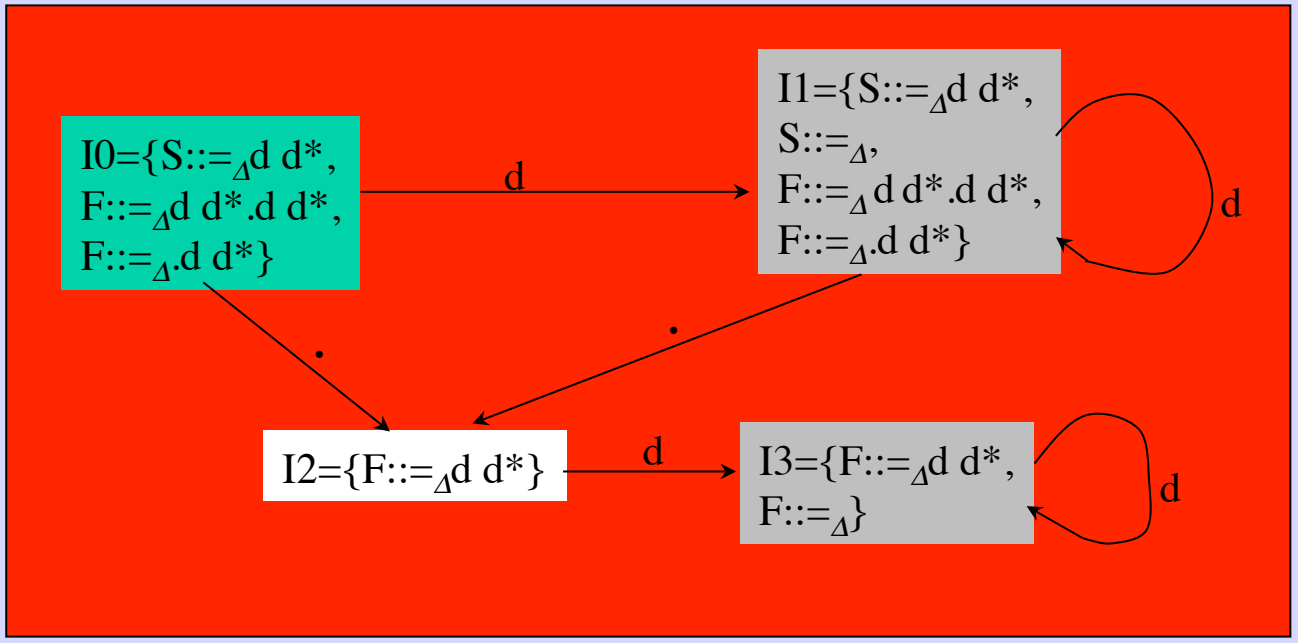
- **Dotted Automata: From RG to DFA**
- let G=<S, Σ, s_0, P> be Regular, and P={s::=e}. It return A_G: L(G)=L(A_G).
  - **Initial State**: $\boldsymbol{I_0}=\boldsymbol{C}(\{s ::=_{\Delta} e \mid s ::= e \in P, s \in S_0 \subseteq S\})$,
    S_0 contains the lexical classes (token) of interest (i.e. no auxiliaries)
  - Definition of the ***Closure Function*** $\boldsymbol{C_s}$ (by induction on *productions*):
    - $\boldsymbol{C_s}(\{s ::=_{\Delta} \beta \mid s \in S\}) = \cup_{s \in S} C(s ::=_{\Delta} \beta)$
    - $C(s ::=_{\Delta} \varepsilon) = \{s ::=_{\Delta}\}$;
    - $C(s ::=_{\Delta} b\beta) = \{s ::=_{\Delta} b\beta\}, \forall b \in \Sigma, \beta \in \boldsymbol{E}_{\Sigma}$;
    - $C(s ::=_{\Delta} s'\beta) = \boldsymbol{C_s}(\{s ::=_{\Delta} e\beta \mid s' ::= e \in P\}) \forall \beta \in \Sigma^*$;
    - $C(s ::=_{\Delta} e^*\beta) = \boldsymbol{C}(s ::=_{\Delta} ee^*\beta) \cup C(s ::=_{\Delta} \beta)$;
    - $C(s ::=_{\Delta} (e1|e2)\beta) = C(s ::=_{\Delta} e1\ \beta) \cup C(s ::=_{\Delta} e2\ \beta)$;
    - *additional cases can be added for optional addional RE operators*
  - Definition of the A_G *move, denoted* **M**, and of the remaining A_G states $\mathbf{I_j}$:
    - **M(Ii**,a)=**Ij sse**
      - a$\in$*first*(***Ii***)=$\{a \mid s ::=_{\Delta} a\beta \in \boldsymbol{Ii}, a \in \Sigma\}$
      - **Ij** = $\boldsymbol{C}(\{s ::=_{\Delta} \beta \mid s ::=_{\Delta} a\beta \in \boldsymbol{Ii}, a \in \Sigma\})$
  - **Final States**: $\{\boldsymbol{Ij} \mid \exists\, sj \in S, sj ::=_{\Delta} \in \boldsymbol{Ij}\}$

Apply it to Grammar

$S ::= A\ B$
$A ::= a$
$B ::= b^*\ b$

Dotted Automata: An Example

S::= digit digit*
F::= digit*.digit digit*



	digit	.
0	1	2
1	1	2
2	3	
3	3	

Minimal Automata

Equivalent States: $\sim$, $\sim_\Pi$

$x \sim y \Leftrightarrow \forall \gamma \in \Sigma^*: (\gamma,x \Rightarrow^* \lambda,x' \Leftrightarrow \gamma,y \Rightarrow^* \lambda,y' \text{ for } x' \sim y')$

$x \sim_\Pi y \Leftrightarrow \forall a \in \Sigma: (a,x \Rightarrow \lambda,x' \Leftrightarrow a,y \Rightarrow \lambda,y' \text{ s.t. } [x']_\Pi = [y']_\Pi)$

Minimization Algorithm

Input: $A=<S, \Sigma, move, s, F>$
Output: $M_A=<S|_{\Pi s}, \Sigma, move|_{\Pi s}, s|_{\Pi s}, F|_{\Pi s}>$

1. $\Pi = \{S \setminus F, F\}$;
2. $\Pi\nu = refine_A(\Pi)$;
3. while $\Pi \neq \Pi\nu$ do{
   1. $\Pi = \Pi\nu$;
   2. $\Pi\nu = refine_A(\Pi)$}
4. $\Pi_S = \Pi$

Apply it to automaton A'

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

Operators:

Let $\Pi=\{P1,\ldots,Pk\}$

$refine_A(\Pi) = \Pi\nu = \{Q1,\ldots,Qh\}$ s.t:

$\forall i \in [1..h]$:

- $\exists\, j \in [1..k]$: $Qi=Pj$, *or*
- $\exists\, j \in [1..k]$, $Qi \subset Pj$ and
  $\forall x,y \in Qi$, $x \sim_\Pi y$ and $\forall z \in Pj \setminus Qi$, $x \not\sim_\Pi z$

Partition of S: $\Pi_S = \{P1,\ldots,Pk\}$

- covering: $S = \cup_{j \in [1..k]} Pj$
- equivalence: $Pi \cap_{i \neq j \in [1..k]} Pj$

Representatives $S|_\Pi$ on S: $\Pi=\{P1,\ldots,Pk\}$

- representative: $[Pi]_\Pi \in Pi$ - $\forall x \in Pi$, $[x]_\Pi=[Pi]_\Pi$
- $S|_\Pi = \{[x]_\Pi | x \in S\}$
- $move|_\Pi = \{<<[s]_\Pi,a>,U|_\Pi> | <<s,a>U> \in move\}$