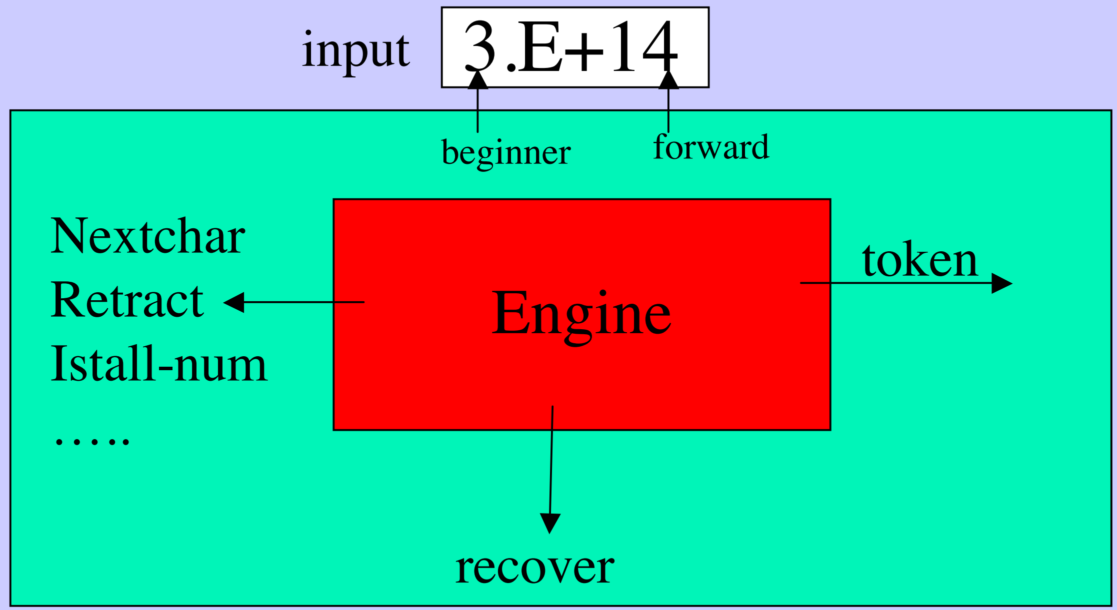
How to Recognize the Lexic defined by a Grammar

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



Scanners do it

AUTOMATA (FSA)
How to build Scanners using FSA as engine

Finite State Automaton:

is a 5-tuple:

$\langle S, \Sigma, \textit{move}: S \times \Sigma' \dashrightarrow S', s_0 \in S, F \subseteq S \rangle$

for a finite set S of *states*, a set Σ of *input values*, a function move of *state transitions*, an *initial state*, a set F of *final states*

NFA (*nondeterministic*)

$$\Sigma' = \Sigma \cup \{\varepsilon\} \quad \text{and} \quad S' = 2^S$$

DFA (*deterministico*)

$$\Sigma' = \Sigma \quad \text{and} \quad S' = S$$

FSA:
Function *move*: Graphs vs. Tables

Graphs [G], strongly similar to *diagrams*, otherwise
Tables [T], for finite functions,
are used to express the finite (**pair set**) function ***move***

To each state s∈S:

G corresponds a distinct vertex of the graph

T corresponds a distict row of with as many colums as cardinality of ∑ (+1 in case of nondeterminism)

To each transition <<*s*,*a*>,*S*>∈*move*

G corresponds putting edge <*s*,*t*>, labeled by *a*, for each *t*∈*S*

T corresponds putting *S* in the entry <*s*,*a*>

FSA:

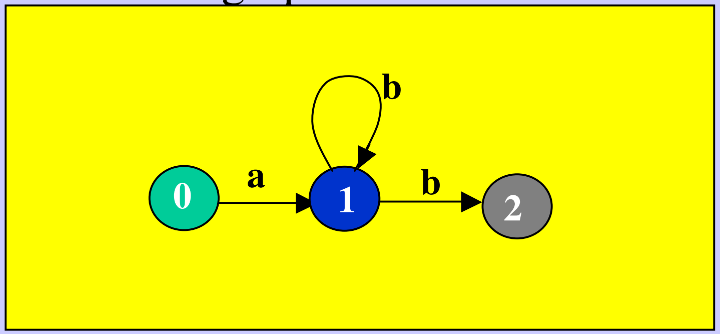
An Example: Graphs vs. Table

A Nondeterministic Finite State Automaton A

Automaton A

<S= {0,1,2},
Σ= {a,b},
move= {<<0,a>{1}>,
<<1,b>{1,2}>}
s_0=0,
F= {2} >

The graph for A’s move



The Table for A’s move

	a	b	ε
0	{1}		
1		{1,2}	
2			

FSA: Meaning

Which *domain* can we use for giving *meaning* to automata to be used as Scanner engine?

A = $<S, \Sigma, \textit{move}: S \times \Sigma' \text{-->} S', s_0 \in S, F \subseteq S>$

Set $\Re \subseteq 2^{\Sigma^*}$ of Regular languages on Σ

FSA
Meaning as *Decision Function* on Σ

To each Automaton we associate a Decision Function f_A

$$\mathrm{sem}(A) = f_A : \Sigma^* \dashrightarrow \{\mathit{accept}, \mathit{noaccept}\}$$

$$\forall\, c_1 c_2 \ldots c_n \in \Sigma^*,$$

$$f_A(c_1 c_2 \ldots c_n) = \begin{cases} \mathit{accept} & \text{if } c_1 c_2 \ldots c_n \in L(A) \\ \mathit{noaccept} & \text{if } c_1 c_2 \ldots c_n \notin L(A) \end{cases}$$

FSA: Decision Function and the binary relation ==> on $(\Sigma^* \times S)^2$

Roughly Speaking

A path, labeled by $\mathbf{c_1 c_2 ... c_n}$, leading from initial state to …, is in the graph …

Formally

Let ==>: $(\Sigma^* \text{ x } S)^2$ be the binary relation defined below

γ, s ==> γ', s' iff

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

or: $\gamma = \gamma'$ and $s' \in$ *move*(ε,s)

Then, Decision Function f is

$$\text{sem}(A)(\gamma) = f(\gamma) = \begin{cases} \textit{accept} & \textbf{if } \gamma, s_0 \text{==>}^* \lambda, s \in F \\ \textit{noaccept} & \textbf{if } \gamma, s_0 \text{=/=>}^* \lambda, s \ (\forall s \in F) \end{cases}$$

Noting the use of the transitive closure ==>* of the relation ==>

FSA:
Language, Equivalence, Minimal

Let A = **<S, Σ, *move*: SxΣ’--> S’, s_0∈S, F⊆S>**

- **Language of A: L**
  - **L(A)** ≡ {γ ∈ Σ* | sem(A)(γ)=accept}
- **Equivalence Set of A: E**
  - **E(A)** ≡ {A’ | sem(A’)=sem(A)}
- **Minimal Automata of A: M**
  - **M(A)** ≡ <S_m, _, _, _,_> ∈ E(A):
    - #S_m ≤ #S for all automata <S, _, _, _,_> ∈ E(A)

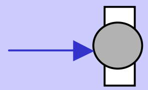
Regular Languages and Automata

- **Each Automaton defines a Regular Language:**
    - proved by the semantics given before and by the 3-step transformation
      Automata *map into* Linear Grammars *map into* Regular Grammars

- (conversely) **Each Regular Language has an Automaton that defines it?**
    - Automata (FSA) ?
        - proved by Thompson's construction
    - Deterministic Automata (DFA) ?
        - proved by the equivalence NFA ≈ DFA ≈ FSA

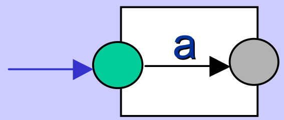
From E_Σ to NFA
K. Thompson's Approach - 1

1. ε



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

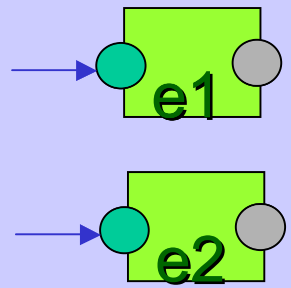
2. $a \quad \forall a \in \Sigma$



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

K. Thompson’ Approach - 2

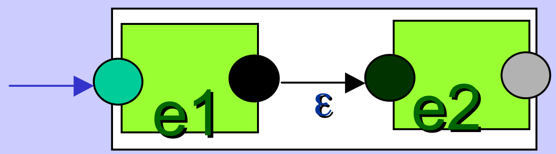
3. e1.e2 $\forall e1, e2 \in E$



e1: $<S_1, \Sigma_1, M_1, s_1, \{f_1\}>$

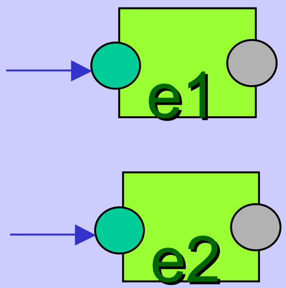
e2: $<S_2, \Sigma_2, M_2, s_2, \{f_2\}>$

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



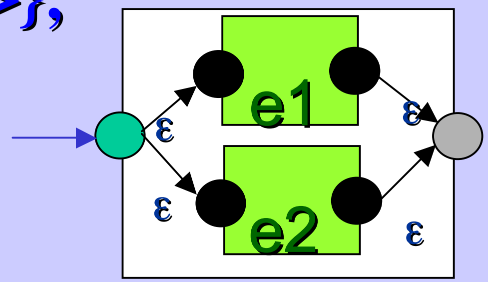
K. Thompson' Approach - 3

4. e1|e2 $\forall$e1,e2$\in$E



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

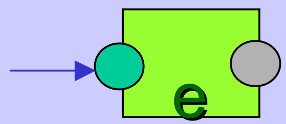
e1|e2: $< S_1 \cup S_2 \cup \{s_{new}, f_{new}\}, \Sigma_1 \cup \Sigma_2,$
$M_1 \cup M_2 \cup \{<< s_{new}, \varepsilon >, \{s_1, s_2\}>,$
$<< f_1, \varepsilon >, \{f_{new}\}>$
$<< f_2, \varepsilon >, \{f_{new}\}>\},$
$s_{new}, \{f_{new}\}>$



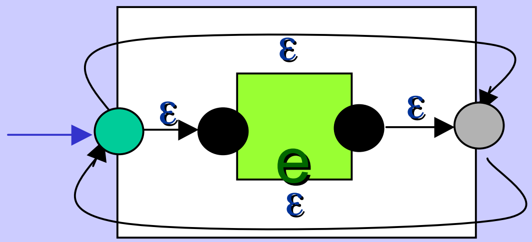
K. Thompson' Approach - 4

5. e* $\forall e \in E$

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

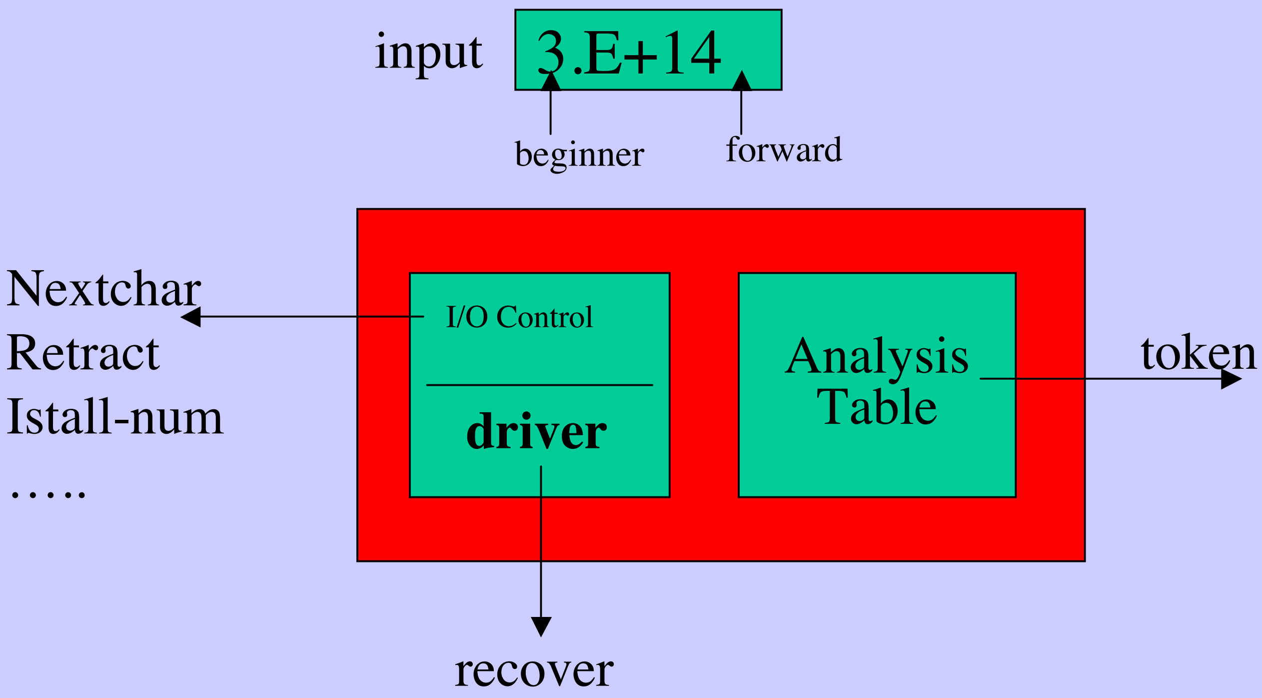


e* : $< S \cup \{s_{new}, f_{new}\}, \Sigma,$
$M \cup \{<< s_{new}, \varepsilon>, \{s, f_{new}\}>,$
$<< f, \varepsilon>, \{s, f_{new}\}>\},$
$s_{new}, \{f_{new}\}>$



The Scanner Core:

A 3-component engine



The idea: **Codify the meaning of a grammar** into the **transition function of an automaton** in order to obtain a table to be used **as Analysis Table** of an engine whose **driver is** (an implementation of) **the automata decision function**.

A Driver for NFA (DFA)

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

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

Answer: A type for {accept,noaccept}
States: control stack
Push: adds[S,&]
S: A set of states
&: Pointer to the current input char
Pop: Removes only 1 state from S, resets input pointer to &. If S is singleton [S,&] is popped from the stack
⊥: result of move[s,c] when the table has no entries for move[s,c]
Clos: ε-closure of a set S of states:
$Clos(S)=S+Clos(\cup_{u \in S} move(u,\varepsilon))$
Nextchar(x): copies in x current input char and reset input pointer to next char

Apply it to automaton A

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

when scanning: aab$

A Driver for NFA (DFA):
How to remove Backtracking - 1

$$f(\gamma) = \begin{cases} \textit{accept} & \textbf{if } \text{move1}^*(\gamma) \cap F \neq \{\} \\ \textit{noaccept} & \textbf{otherwise} \end{cases}$$

move

	a	b	ε
0	{1}		
1		{1,2}	
2			

move1

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

How to remove Backtracking

Function *move1**: Σ^* -> S

How to computes transitions using Set of States instead of single States

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

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

Then:

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

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

Then, Decision Function f can be reformulated in

$$sem(A)(\gamma) = f(\gamma) = \begin{cases} accept & \textbf{if } move1^*(\gamma) \in F \\ noaccept & \textbf{if } move1^*(\gamma) \notin F \end{cases}$$

How to remove Backtracking

A linear Driver for NFA/DFA

The implementation of ***move1**** leads to a new linear, deterministic, recogniser

```
Answer move1star()
        {S= Clos({s0})};
         nextchar(c);
         while(c≠eof) and (S≠∅) {
                 S=move1(S,c);
                 nextchar(c)};
         if (S∩F) ≠ ∅  return 'accept';
         return 'noaccept';
        }
```

Linear *(but step move1 requires exponential time)*

Apply it to automaton A

<**S**= {0,1,2}, **Σ**= {a,b},
move= {<<0,a>{1}>,
<<1,b>{1,2}>}
s_0=0, **F**= {2} >

when scanning: abb$