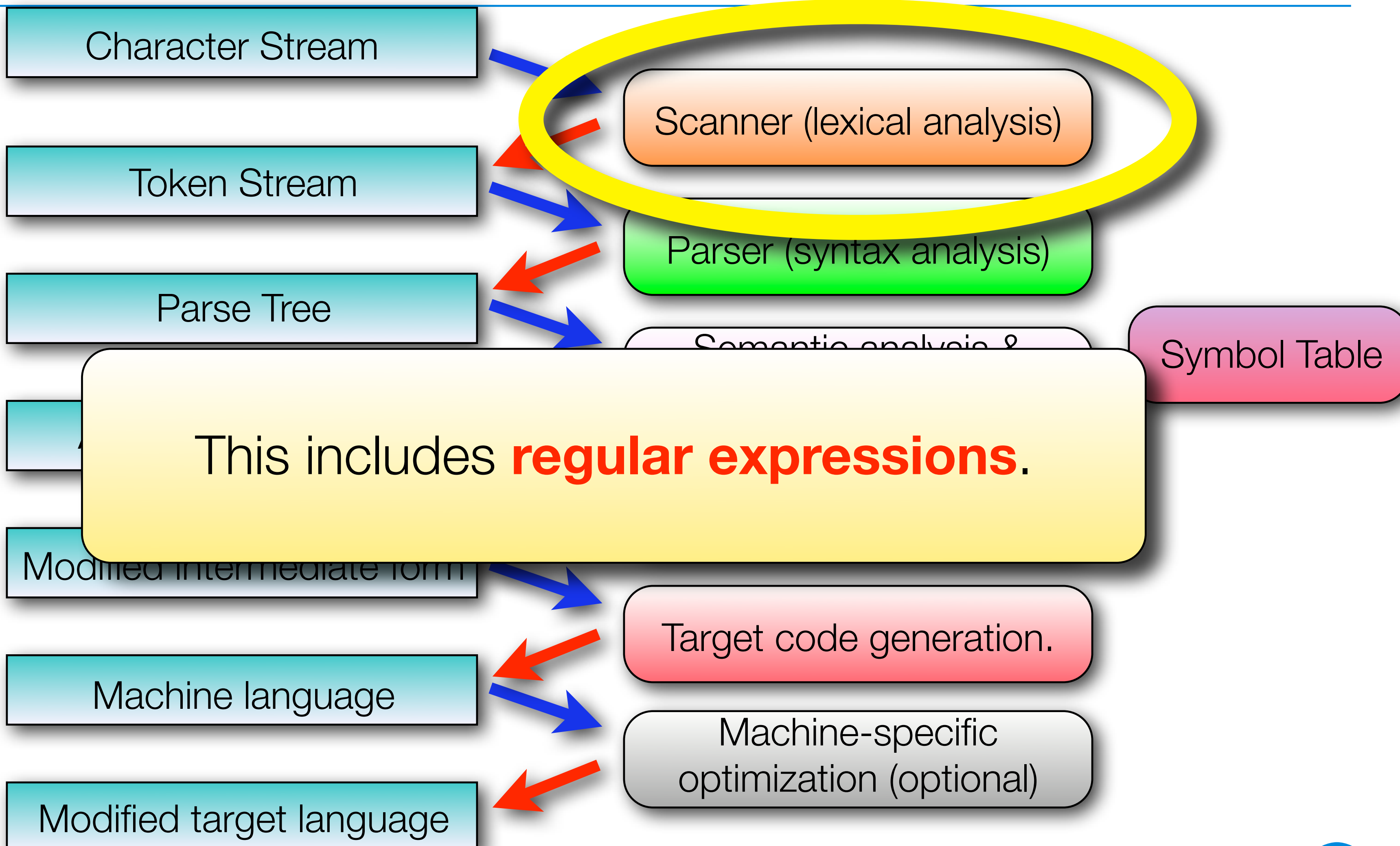
Lecture 3: Lexical Analysis

COMP 524 Programming Language Concepts
Stephen Olivier
January 20, 2009

Based on notes by A. Block, N. Fisher, F. Hernandez-Campos, J. Prins and D. Stotts

The University of North Carolina at Chapel Hill

Goal of Lecture



This includes **regular expressions**.

Scanning

- The main task of scanning is to **identify tokens**.

Pseudo-Code Scanner (Fig 2.5)

```
We skip any initial white spaces
we read the next character
if it is a ( we look at the next character
  if that is a * we have a comment;
    we skip forward through the terminating *)
  otherwise we return a ( and reuse the look-ahead
If it is one of the one-character tokens ([],;=+- etc.)
  we return that token
...
```

Pseudo-Code Scanner (Fig 2.5)

```
We skip any initial white spaces
we read the next character
if it is
  if that
    we sk
  otherwi
If it is                                                        .)
  we return that token
...
```

We could just turn this into real code and use that as the scanner, and that would be fine for small programs...

Pseudo-Code Scanner (Fig 2.5)

```
We skip any initial white spaces
we read the next character
if it is
  if that
    we sk
  otherwi
If it is                                              .)
  we return that token
...
```

... However, for larger programs that must be correct a more formal approach is more appropriate.

Regular expressions

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit* non_zero_digit*) | ε)

Regular expressions

digit → 0 | | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digi 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non | ε)

"→" denotes **assignment**

Regular expressions

digi → 0 | 1 | | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digi | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *on_zero_digit digit**

non | ε)

"|" denotes **or**

Regular expressions

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

non_zero_digi 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non | ε)

Thus, digit equal “0” or “1” or “2” or ....

Regular expressions

"*****" denotes **zero or more of this type**.

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit** *non_zero_digit*) | ε)

Regular expressions

Two REs next to each other denotes **concatenation**.

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit* non_zero_digit*) | ε)

Regular expressions

So natural number equals at least
"one non-zero digit" followed by "zero or more digits".

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit* non_zero_digit*) | ε)

Regular expressions

“**ε**” means **empty**.

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit** *non_zero_digit*) | ε)

Regular expressions

So, what does this mean?

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit** *non_zero_digit*) | ε)

Regular expressions

It means "**0** or a **natural number**" followed by "**nothing**" or by "**.** and **zero or more digits** concluded by a **non_zero number**"

non_zero_digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

natural_number → *non_zero_digit digit**

non_neg_number → (0 | *natural_number*) ((. *digit** *non_zero_digit*) | ε)

Regular Expression Rules

- A RE consist of:
  - A character (e.g., “0”, “1”, ...)
  - The empty string (i.e., “ε”)
  - Two REs next to each other (e.g., “*non_negative_digit digit*”) to denote concatenation.
  - Two REs separated by “|” next to each other (e.g., “*non_negative_digit* | *digit*”) to denote one RE or the other.
  - An RE followed by “*” (called the **Kleene star**) to denote zero or more iterations of the RE.
  - Parentheses (in order to avoid ambiguity).

Regular Expression Rules

- A RE consist of:
  - A character (e.g., "0", "1", ...)
  - The (
  - Two ... *digit*") to deno
  - Two REs separated by "|" next to each other (e.g., "*non_negative_digit* | *digit*") to denote one RE or the other.
  - An RE followed by "*" (called the **Kleene star**) to denote zero or more iterations of the RE.
  - Parentheses (in order to avoid ambiguity).

A RE is **NEVER** defined in terms of itself! Thus, REs cannot **define recursive statements**.

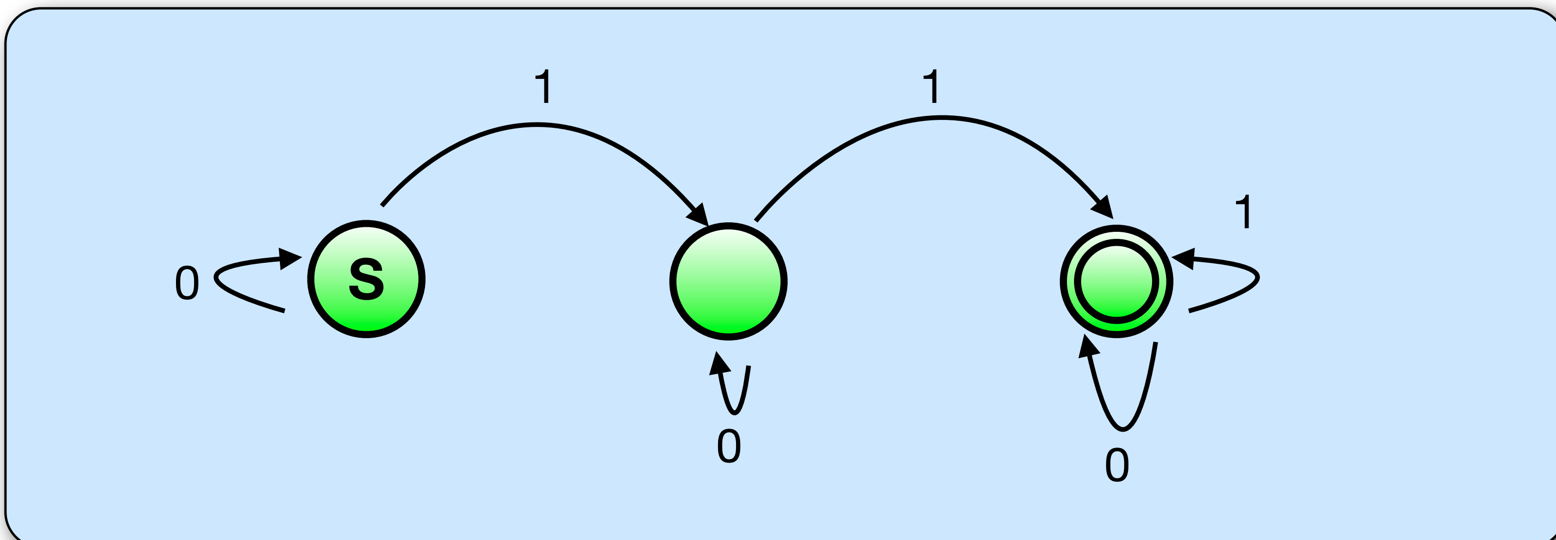
Regular Expression Rules

- A RE consist of:
  - A character (e.g., “0”, “1”, ...)
  - The
  - Two *igit*”) to deno
  - Two REs separated by “|” next to each other (e.g., “*non_negative_digit* | *digit*”) to denote one RE or the other.
  - An RE followed by “*” (called the **Kleene star**) to denote zero or more iterations of the RE.
  - Parentheses (in order to avoid ambiguity).

The set of tokens that can be recognized by regular expressions is called a **regular set**.

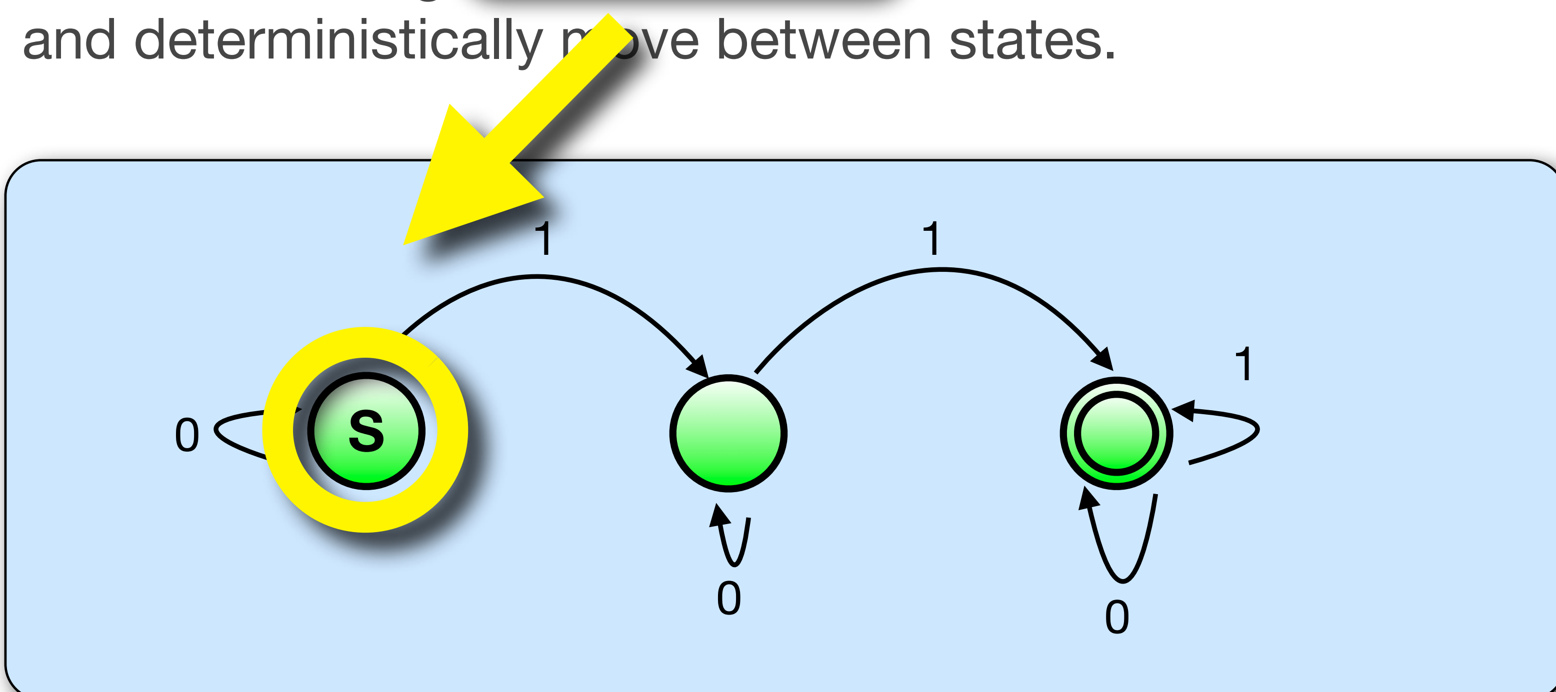
Deterministic finite automaton (DFA)

- Every regular set can be defined by using **deterministic finite automaton** (**DFA**).
  - DFAs are turing machines that have a finite number of states and deterministically move between states.



Deterministic finite automaton (DFA)

- Every regular set can be defined by using **deterministic finite automato**
  - DFAs are turning machines that have a finite number of states and deterministically move between states.

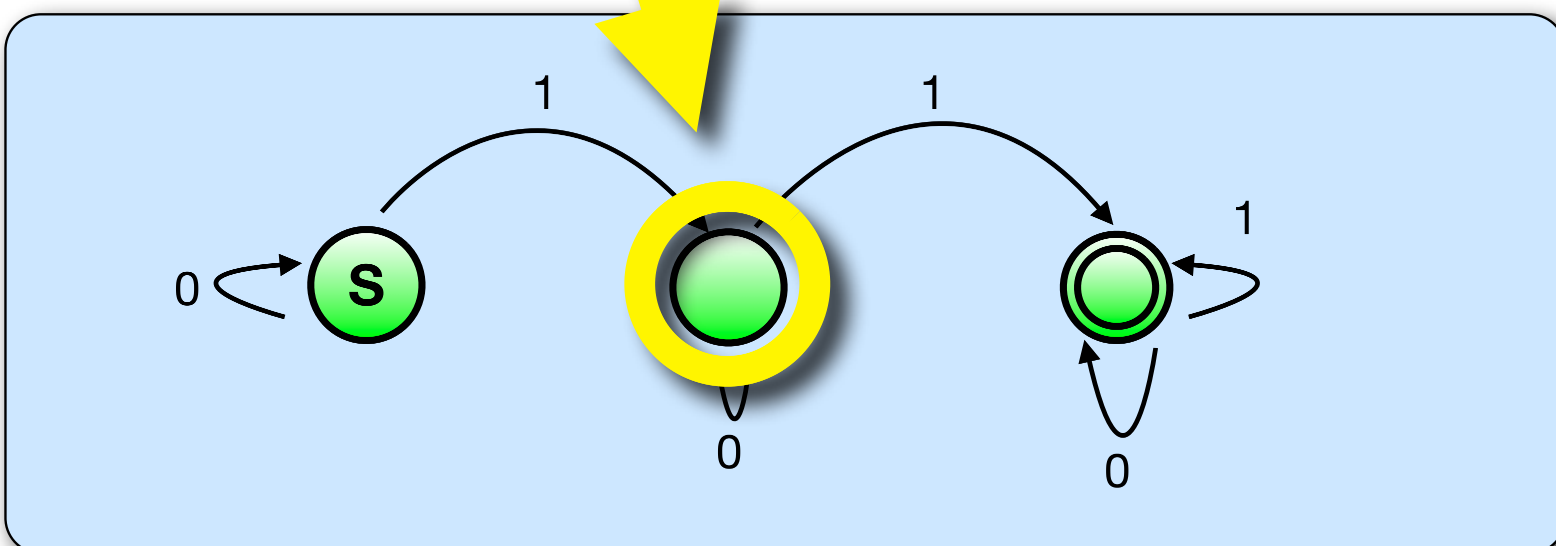


Deterministic finite automaton (DFA)

- Every regular set can be defined by using **deterministic finite auto**

Intermediate State

  - DFAs are turning machines that have a finite number of states and deterministically m ve between states.

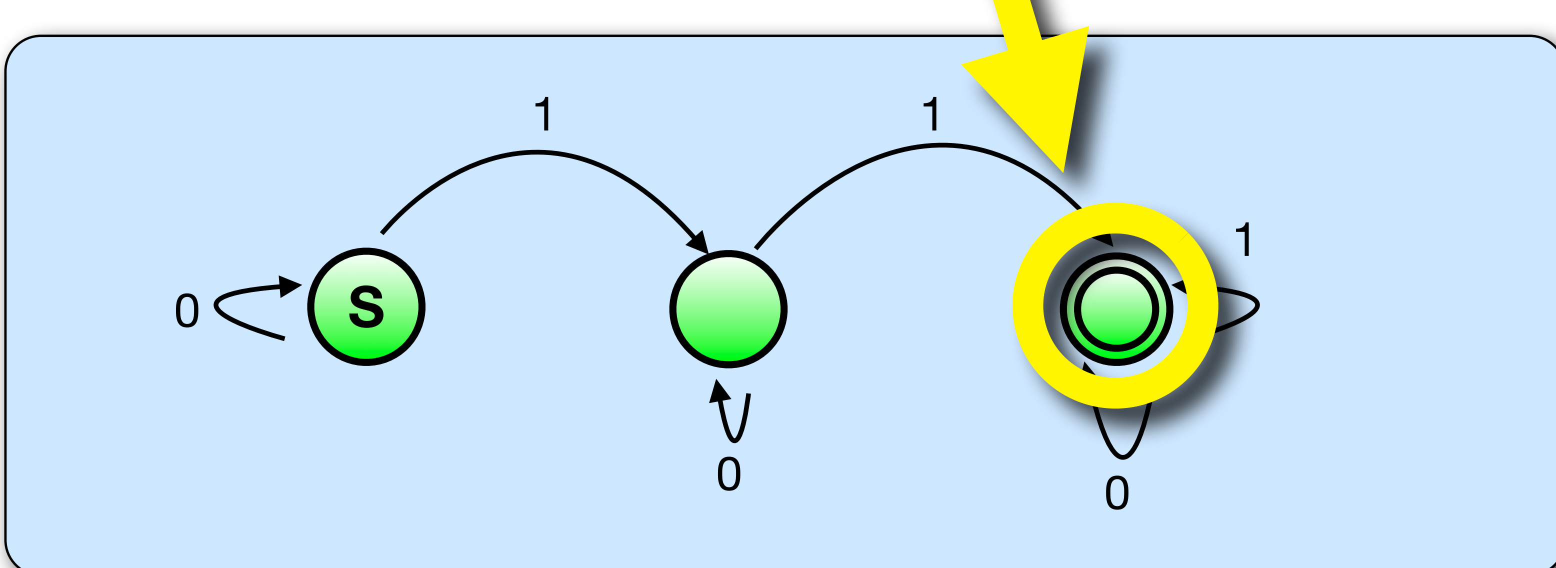


Deterministic finite automaton (DFA)

- Every regular set can be defined by using **deterministic finite automaton** (

End State (double circle)

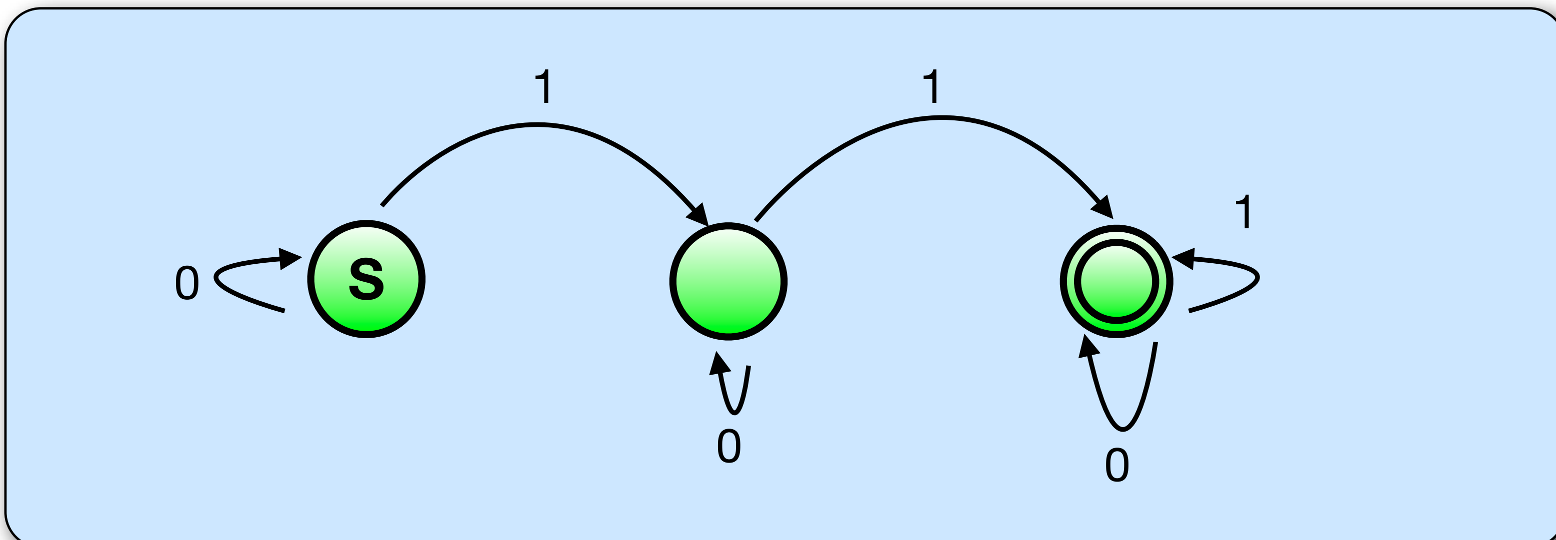
  - DFAs are turning ma… r of states and deterministically move between states.



Deterministic finite automaton (DFA)

- Every regular set can be defined by using **deterministic finite a**
  - DFAs a … number of states and de

This stands for:
0*10*1(0|1)*

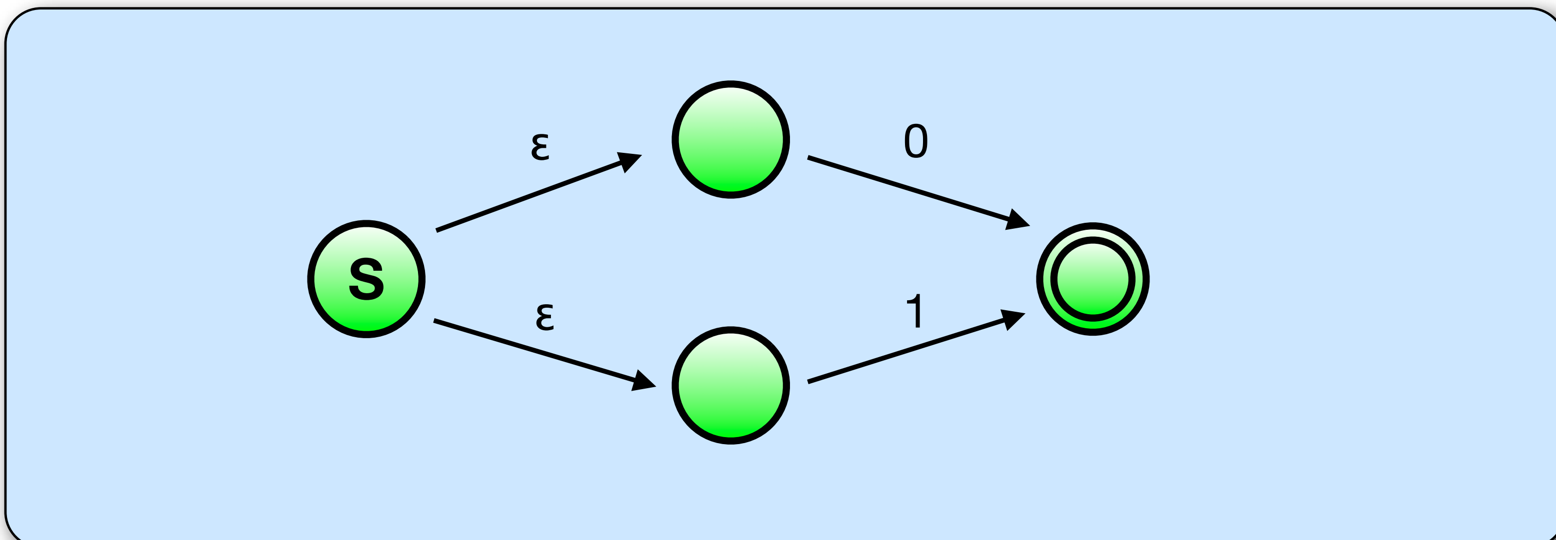


Constructing DFAs

- A DFA can be constructed from a RE via two steps.
  1. Construct a **nondeterministic finite automaton** (**NFA**) from the RE.
  2. Construct a DFA from the NFA.
  3. Minimize the DFA

What is an NFA?

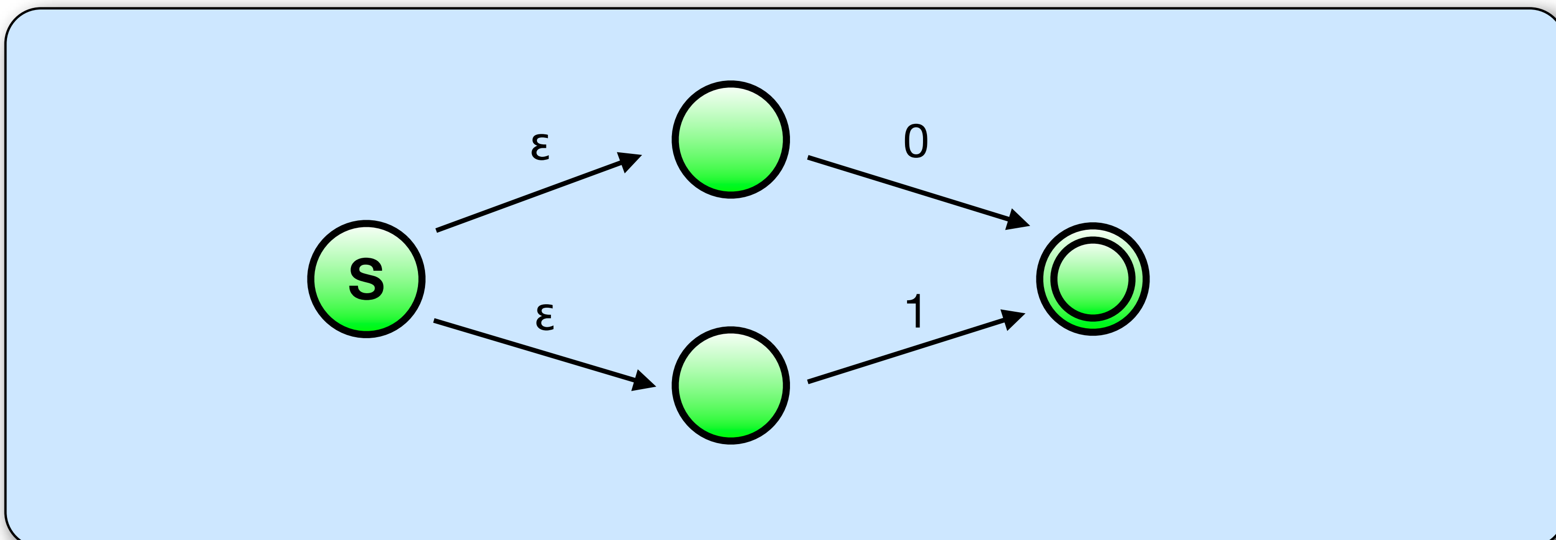
- An NFA is similar to a DFA, except that state transitions are nondeterministic.
- This nondeterminism is encapsulated via the **epsilon transition** (written as **ε**).



What is an NFA?

- An NFA is similar … that state transitions are nondeterminis…
- This nondetermin… ed via the **epsilon transition** (writte…

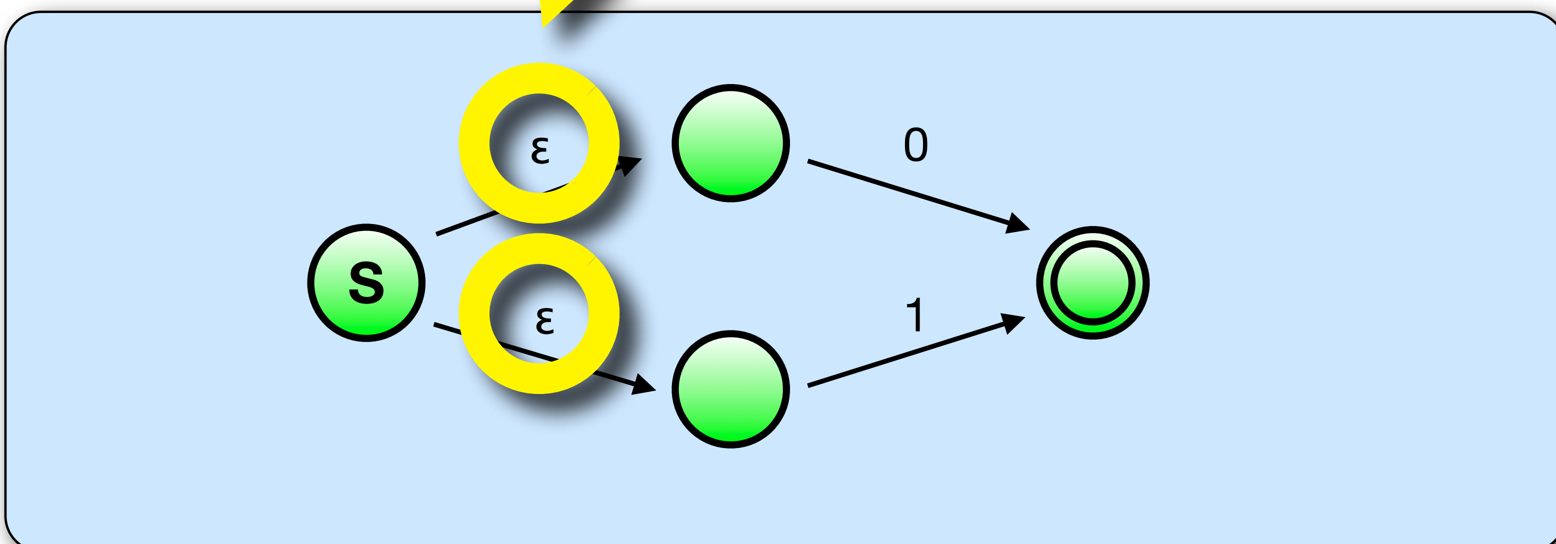
This stands for:
0|1



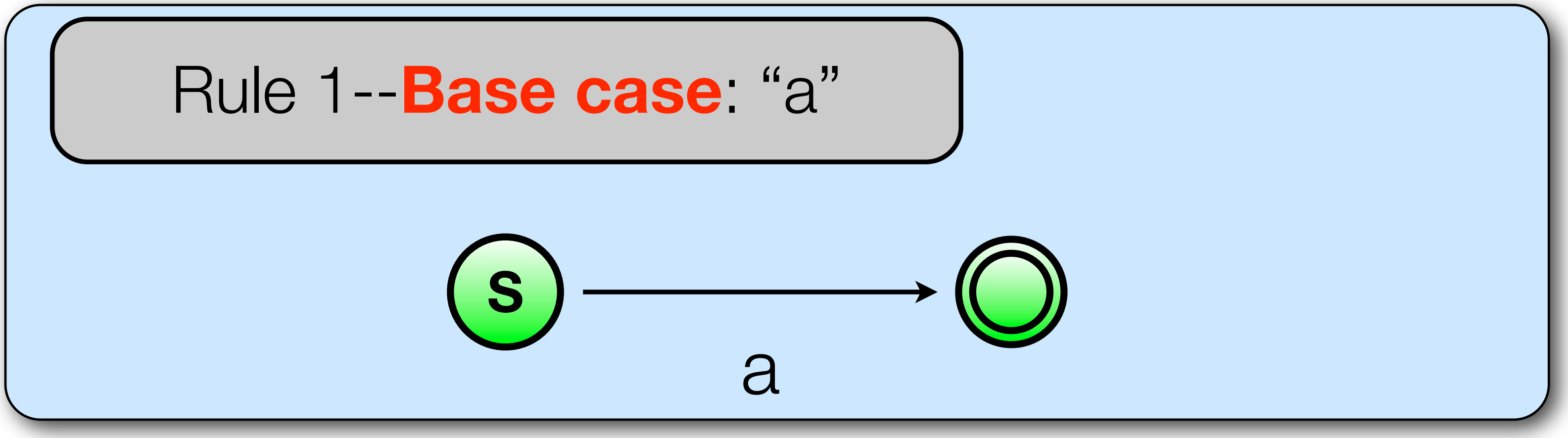
What is an NFA?

The **ε** transitions imply that either transition can be taken with any (or no) input.

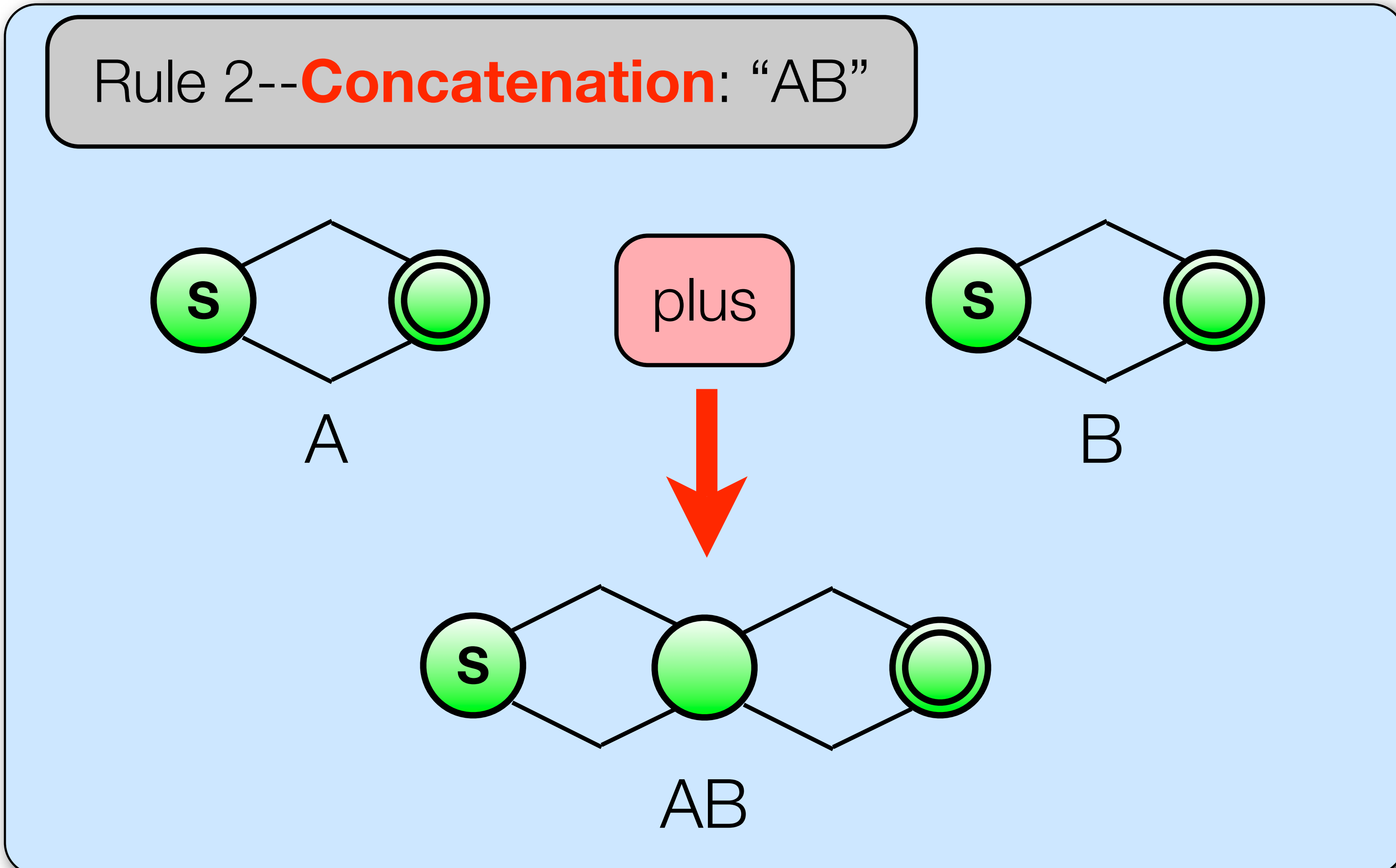
- An NF ate transitions are nc
- This nondeterminism is encapsulated via the **epsilon transition** (written as **ε**).



The four RE rules and NFA



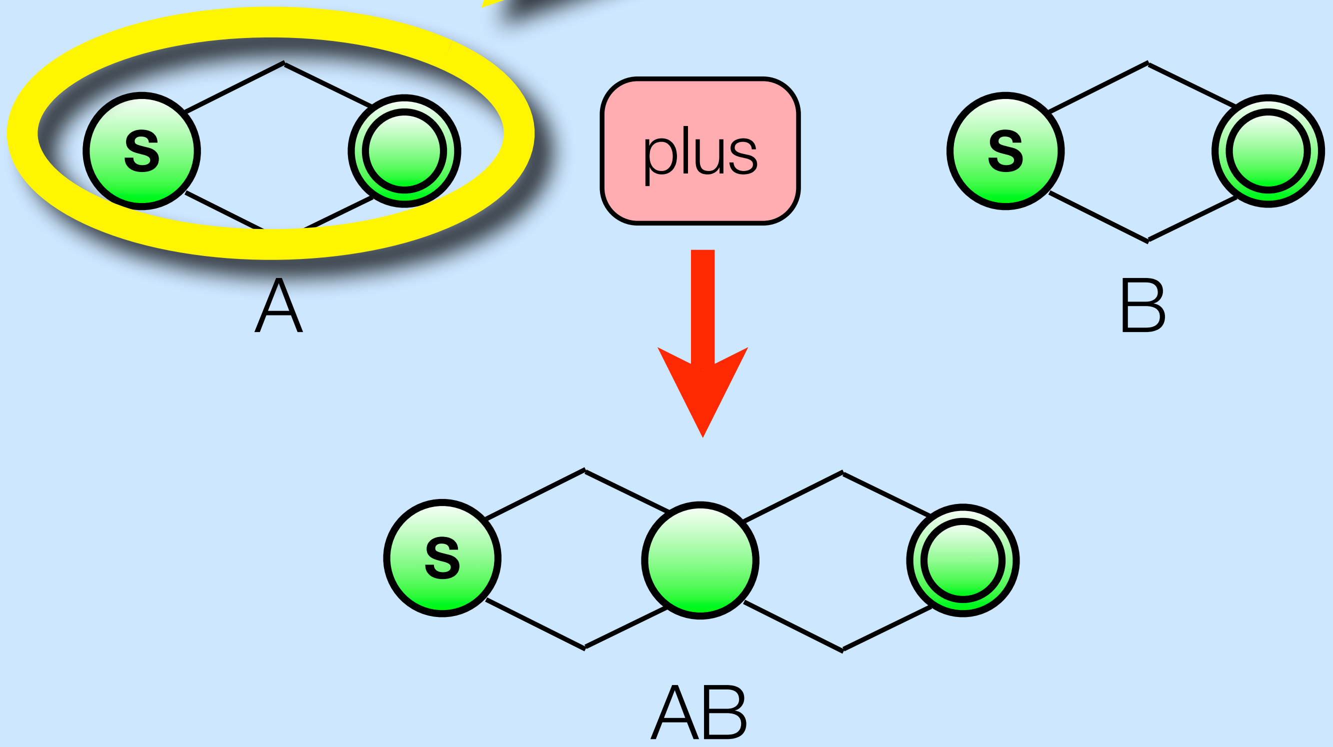
The four RE rules and NFA



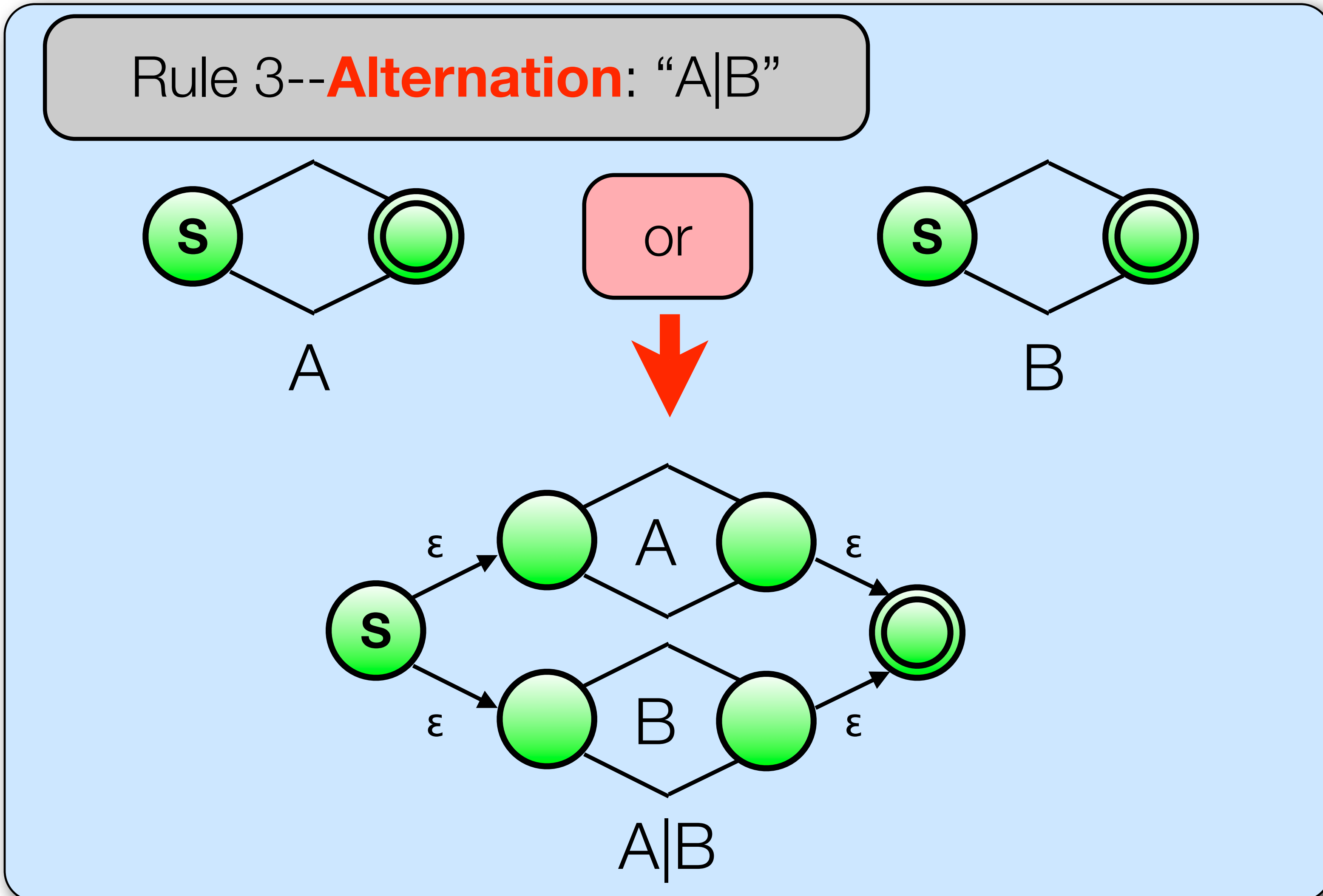
The fo

Stands for Some NFA called “A”

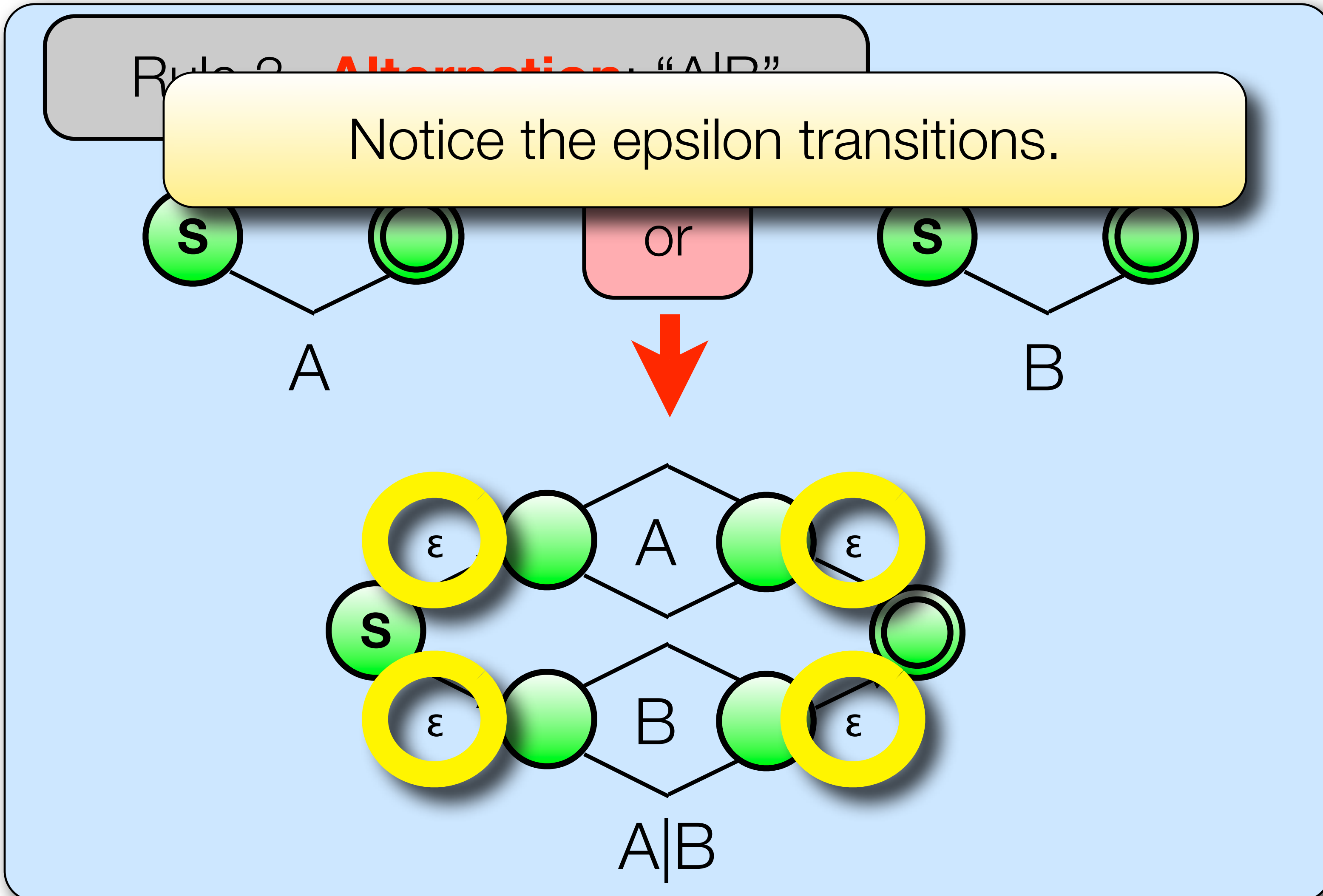
Rule 2--**Concate tion**: “AB”



The four RE rules and NFA

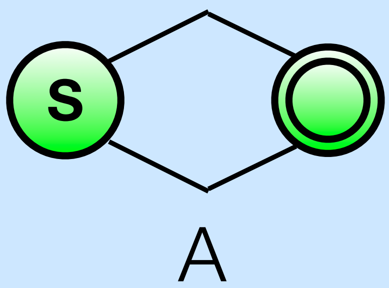


The four RE rules and NFA

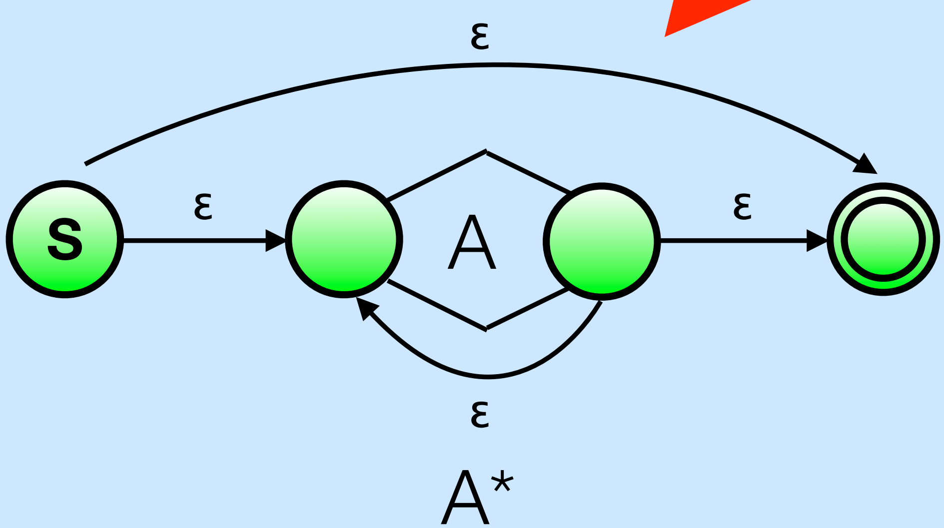


The four RE rules and NFA

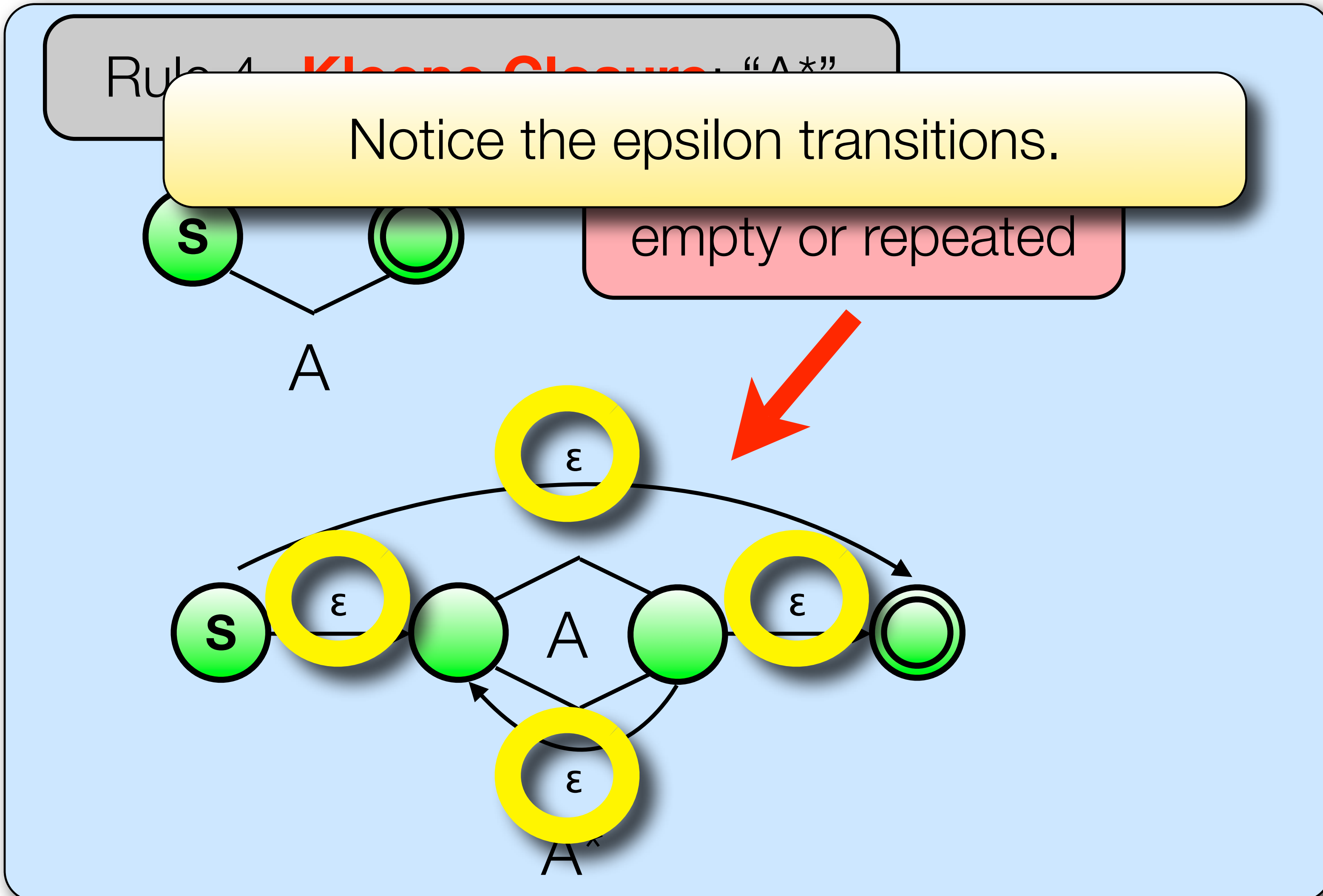
Rule 4--**Kleene Closure**: “A*”



empty or repeated

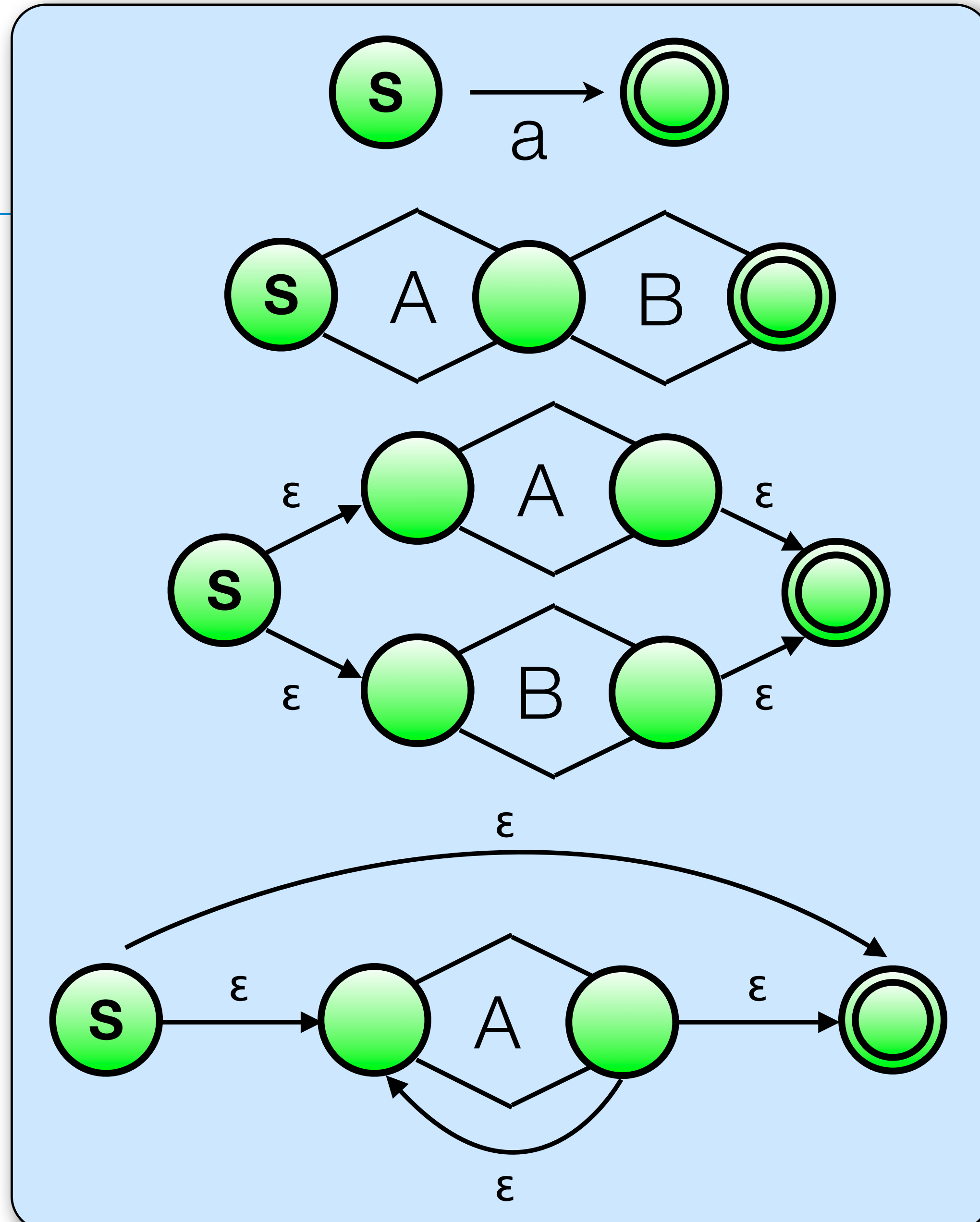


The four RE rules and NFA



Some examples:

- 0|1*
- AB*
- F|(GH*)
- Z*|ε|Y*X



Constructing DFAs

- A DFA can be constructed from a RE via two steps.
  1. Construct a **nondeterministic finite automaton** (**NFA**) from the RE.
  2. Construct a DFA from the NFA.
  3. Minimize the DFA

Constructing DFAs

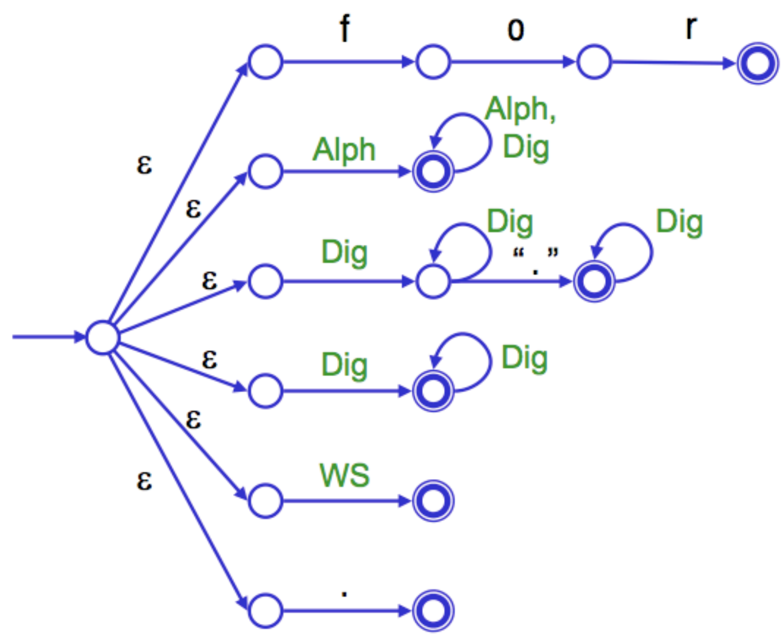
- A DFA can be constructed from a RE via two steps.
  1. Construct a **nondeterministic finite automaton** (**NFA**) from the RE.

FOR_KEYWORD → for

IDENTIFIER → Alph (Alph | Dig)*

REAL → *Dig Dig* . Dig**

INT → *Dig Dig**



Constructing DFAs

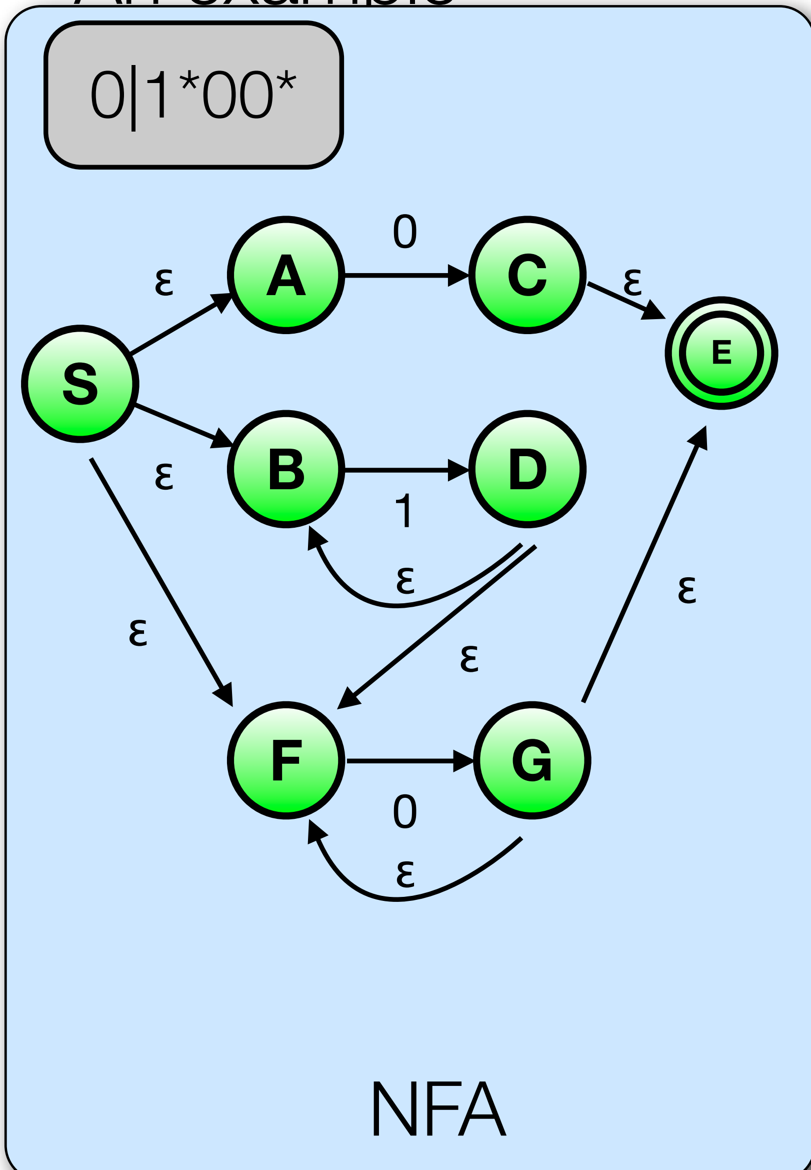
- A DFA can be constructed from a RE via two steps.
   1. Construct a **nondeterministic finite automaton** (**NFA**) from the RE.
   2. Construct a DFA from the NFA.
   3. Minimize the DFA

Constructing a DFA from an NFA.

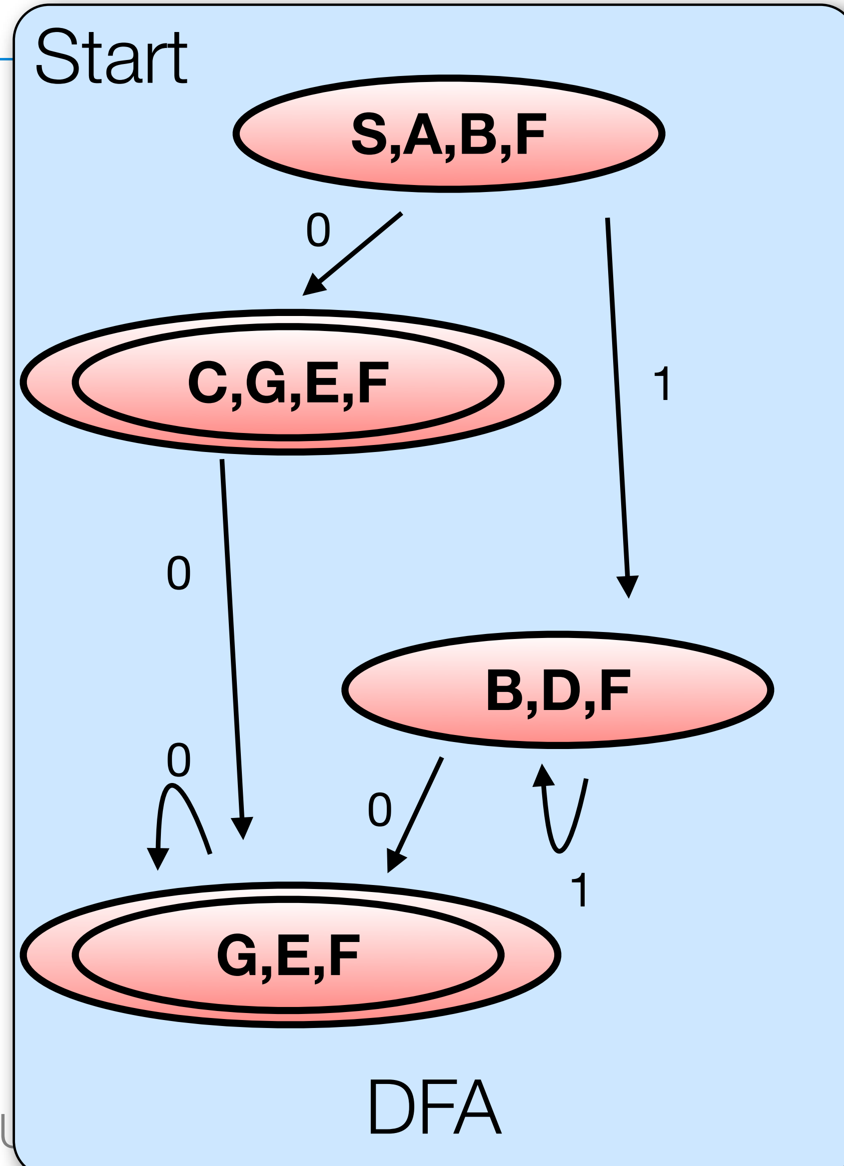
- Construct the DFA by “collapsing” the states of an NFA.
- Three steps

1. Identify set of states that can be reached from the **start state** via **epsilon-transitions** and make this **one state**.
2. For a given DFA state (which is a set of NFA states) consider each possible input and combine the **resulting NFA states into one DFA state**.
3. Repeat Step 2 until all states have been added.

An example



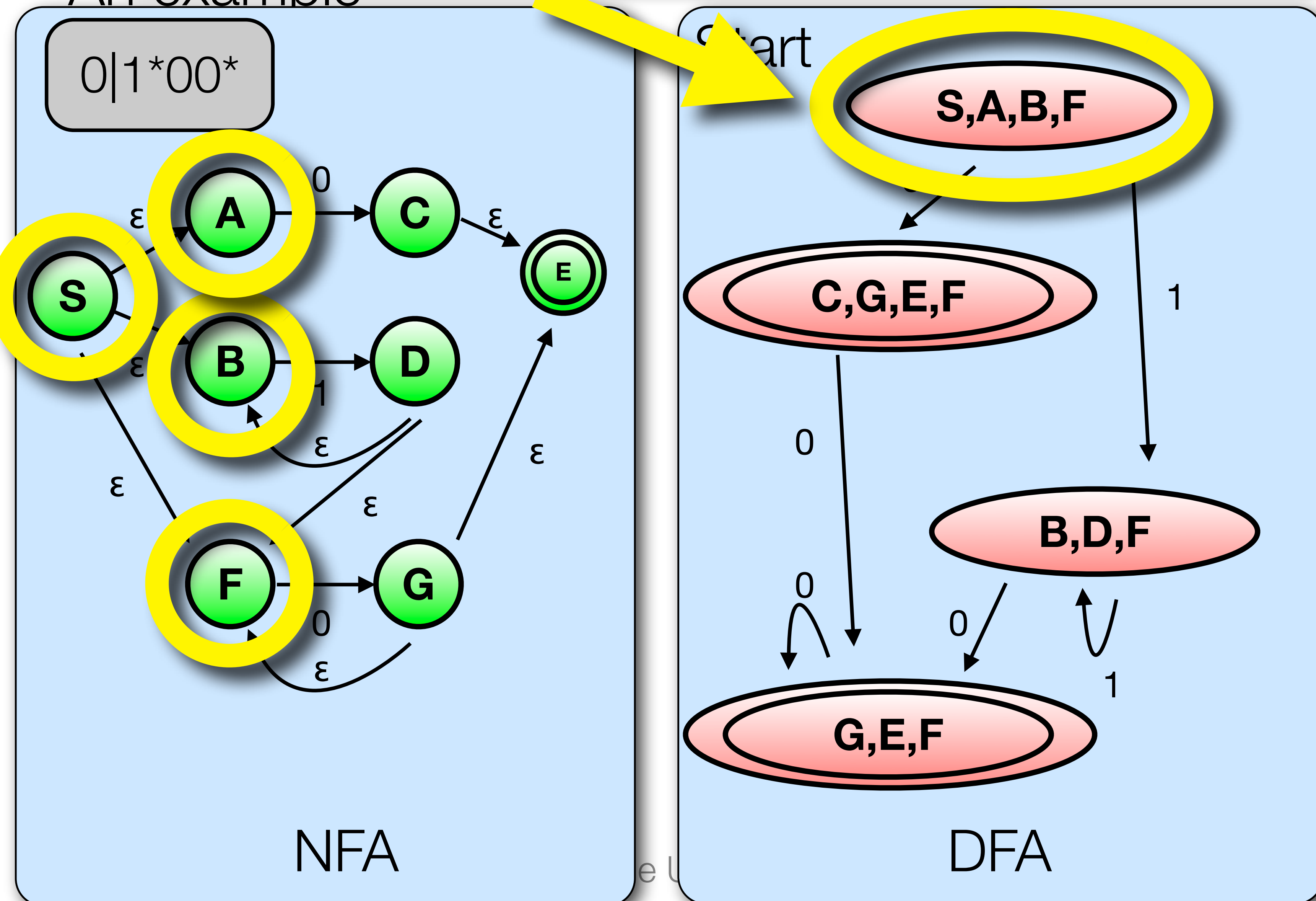
0|1*00*
ε
0
ε
S
A
C
E
B
D
1
ε
ε
ε
ε
F
G
0
ε
NFA



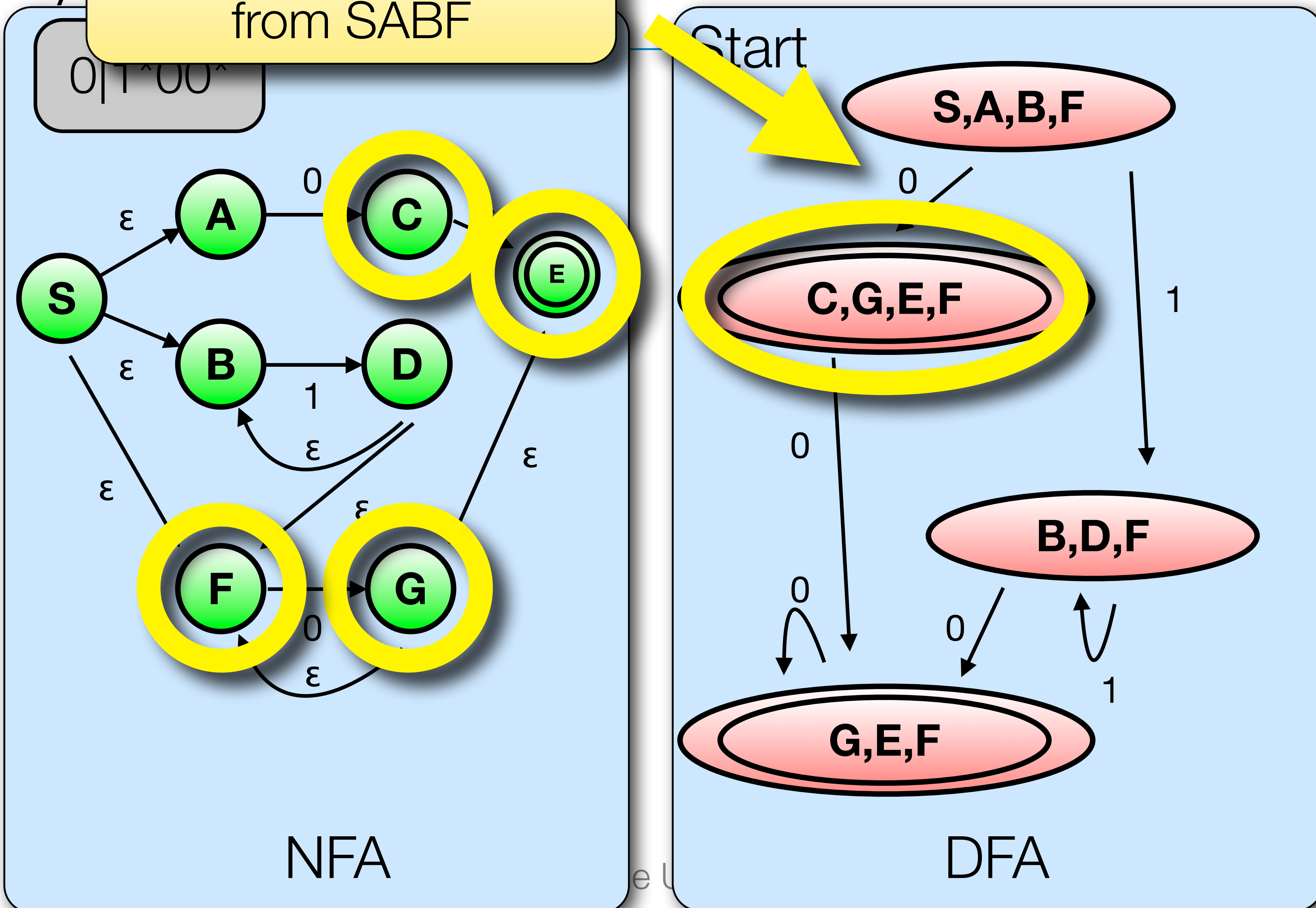
Start
S,A,B,F
0
1
C,G,E,F
0
B,D,F
0
0
1
G,E,F
DFA

All the states that we can reach via **ε** are in this state

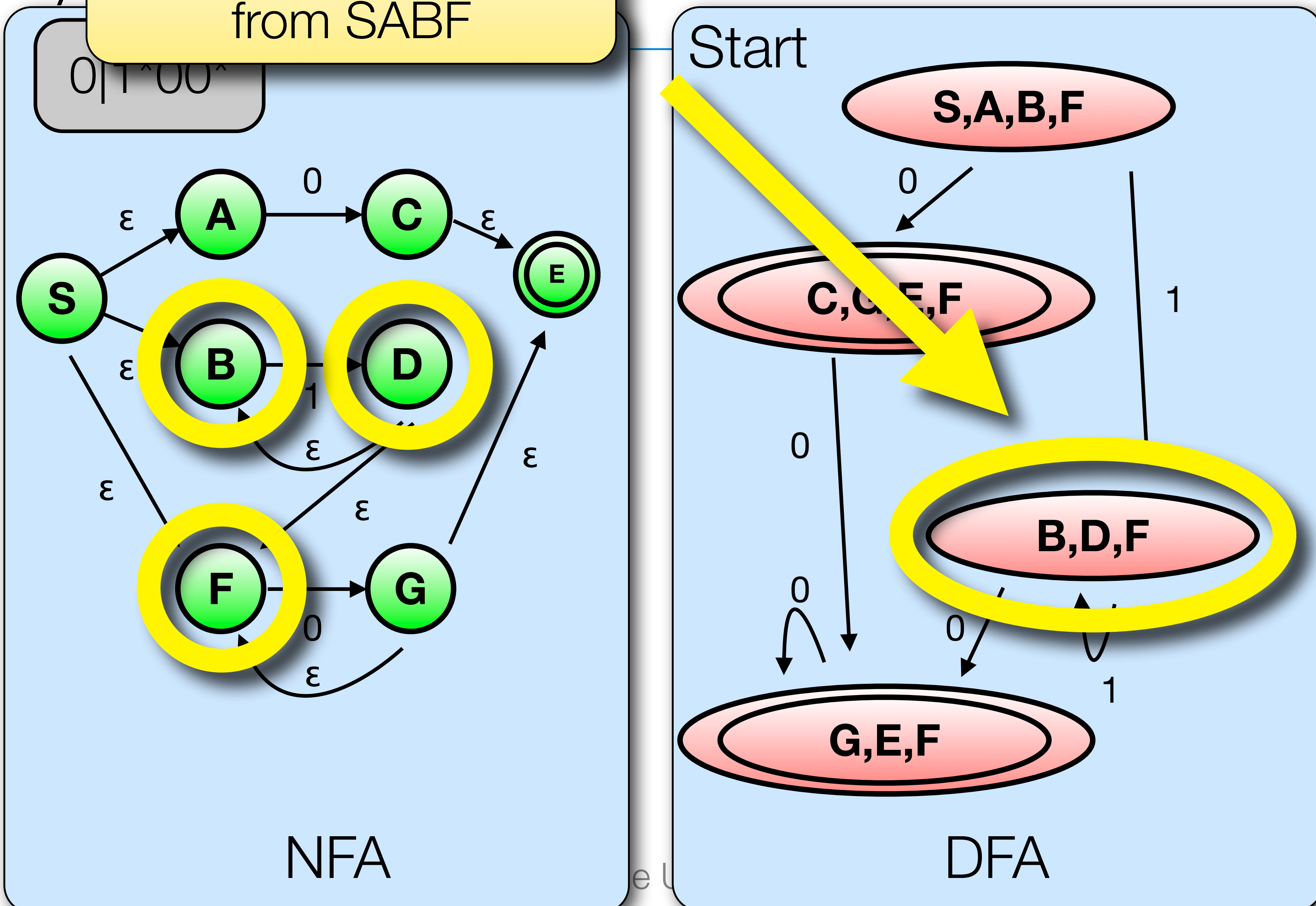
An example

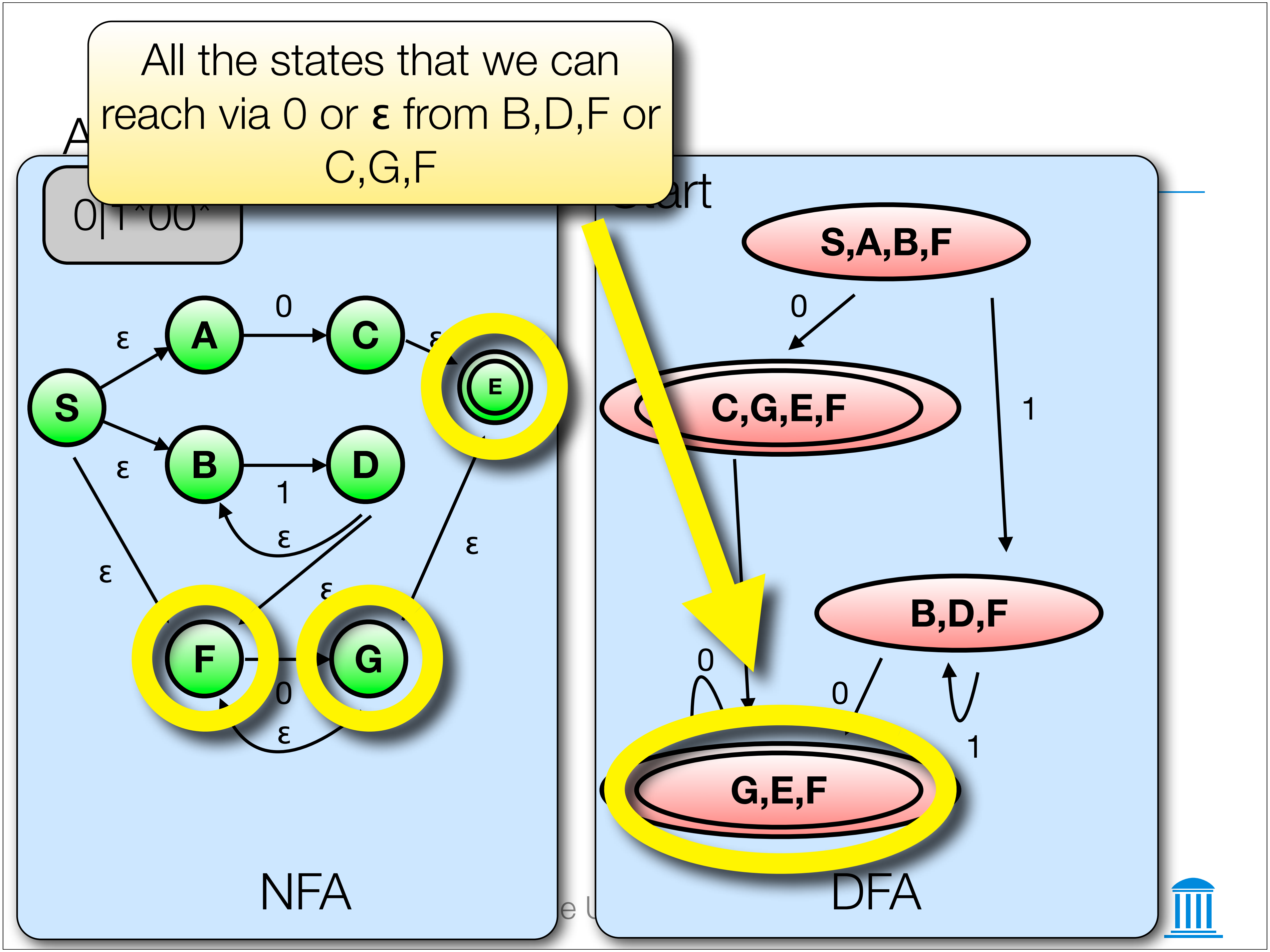


All the states that we can reach via 0 or **ε** from SABF

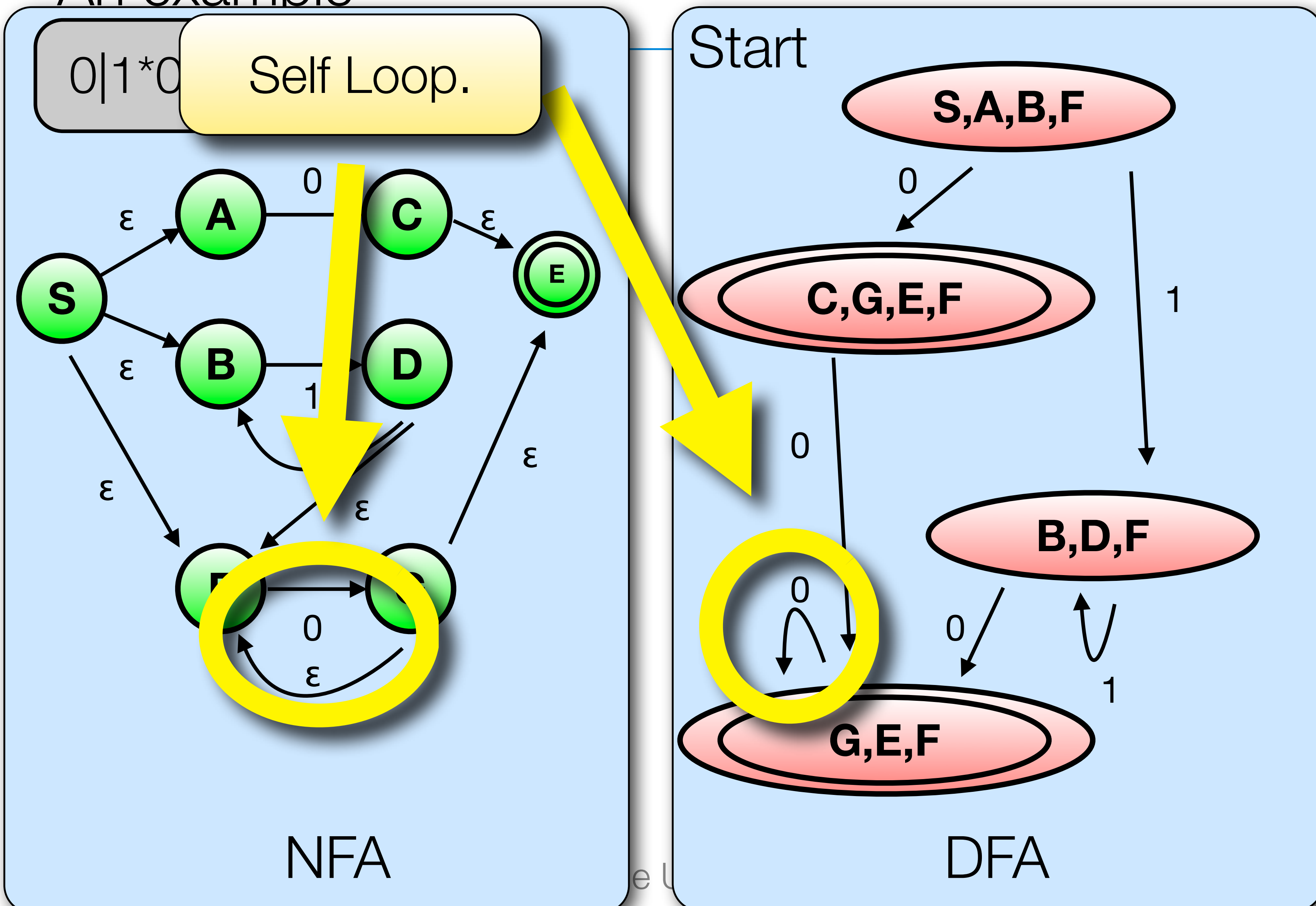


All the states that we can reach via 1 or **ε** from SABF

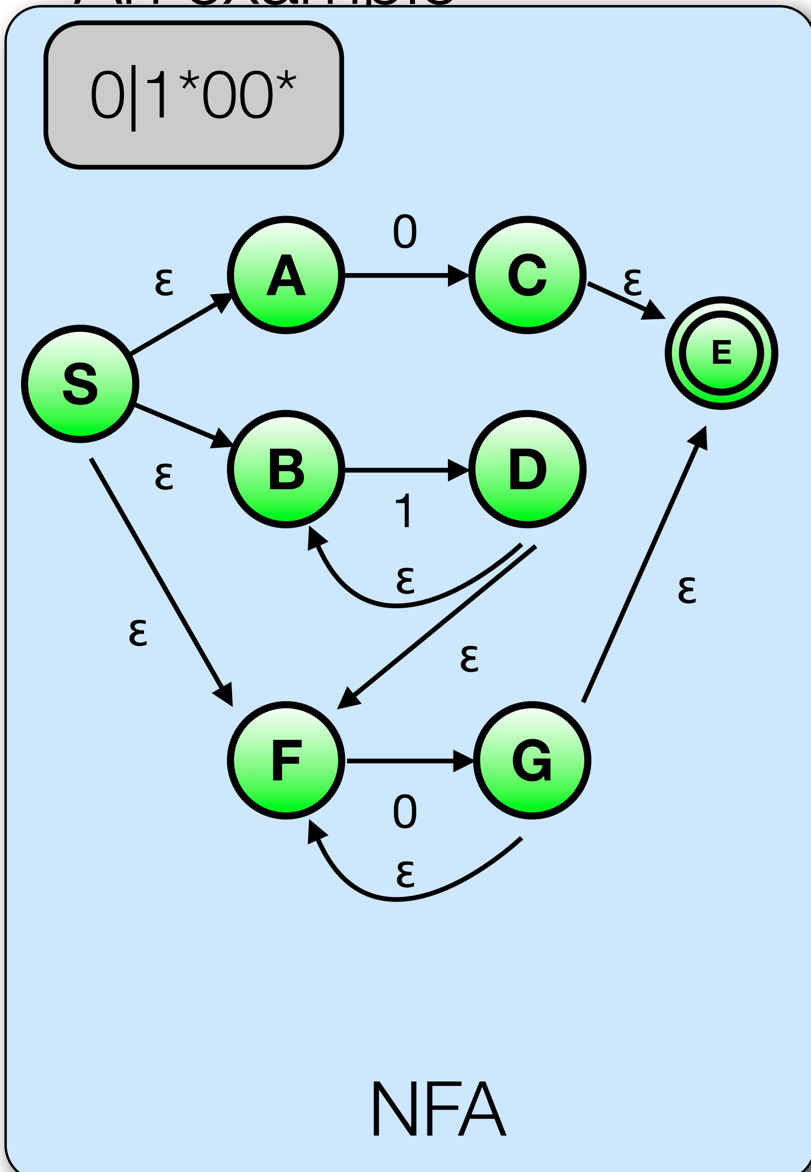


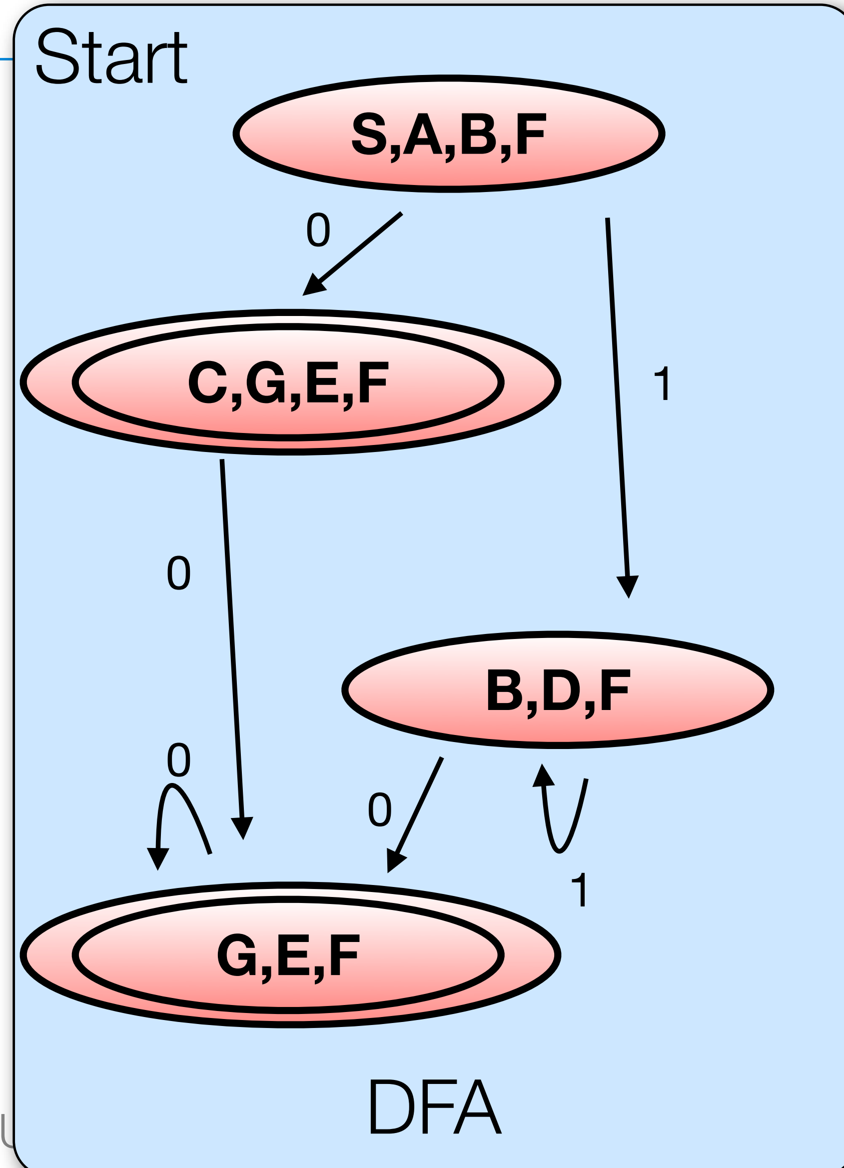

All the states that we can reach via 0 or ε from B,D,F or C,G,F
0|1*00*
S
A
B
C
D
E
F
G
ε
0
1
NFA
S,A,B,F
C,G,E,F
B,D,F
G,E,F
DFA

An example



An example



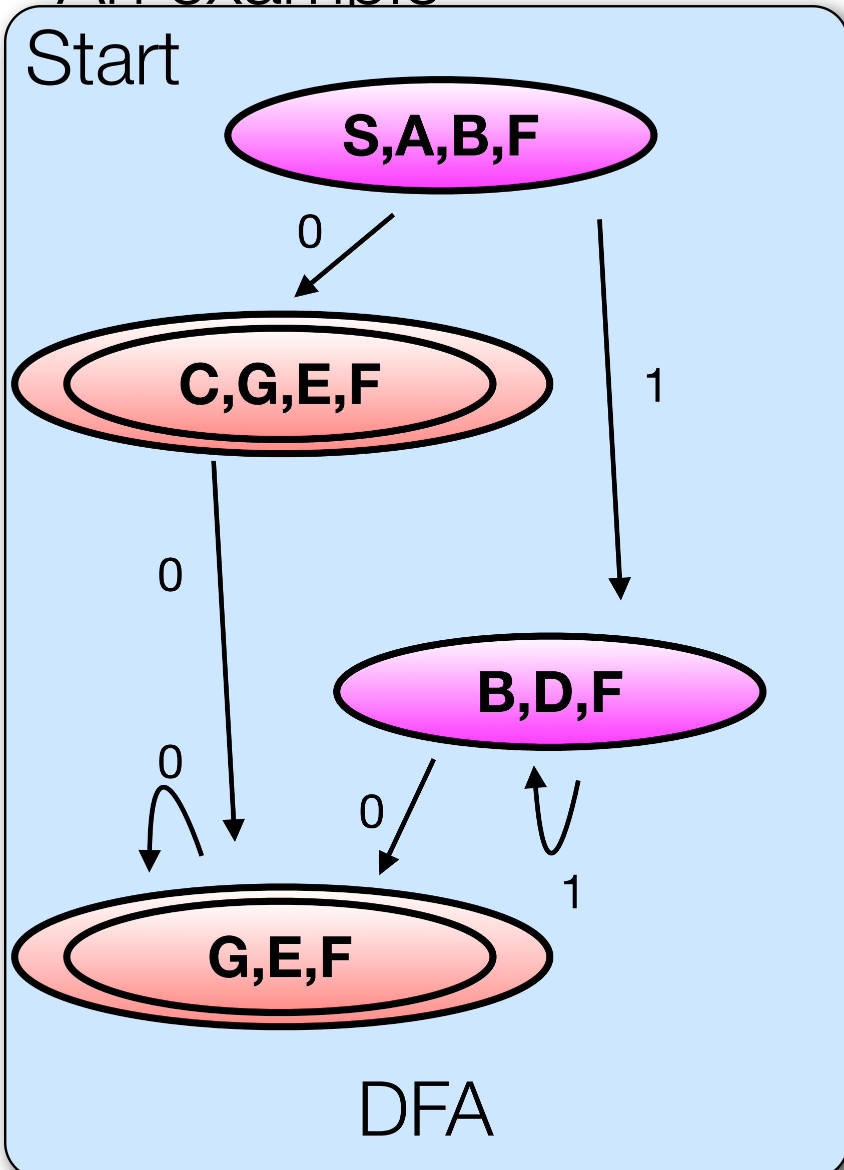


Minimize via partitioning

- First, partition states into final and non-final
- Second, determine the effect of the state transition based on what partition the transition goes to.
- Third, Create new partition for those states that have different transitions.
- Fourth, repeat.

An example

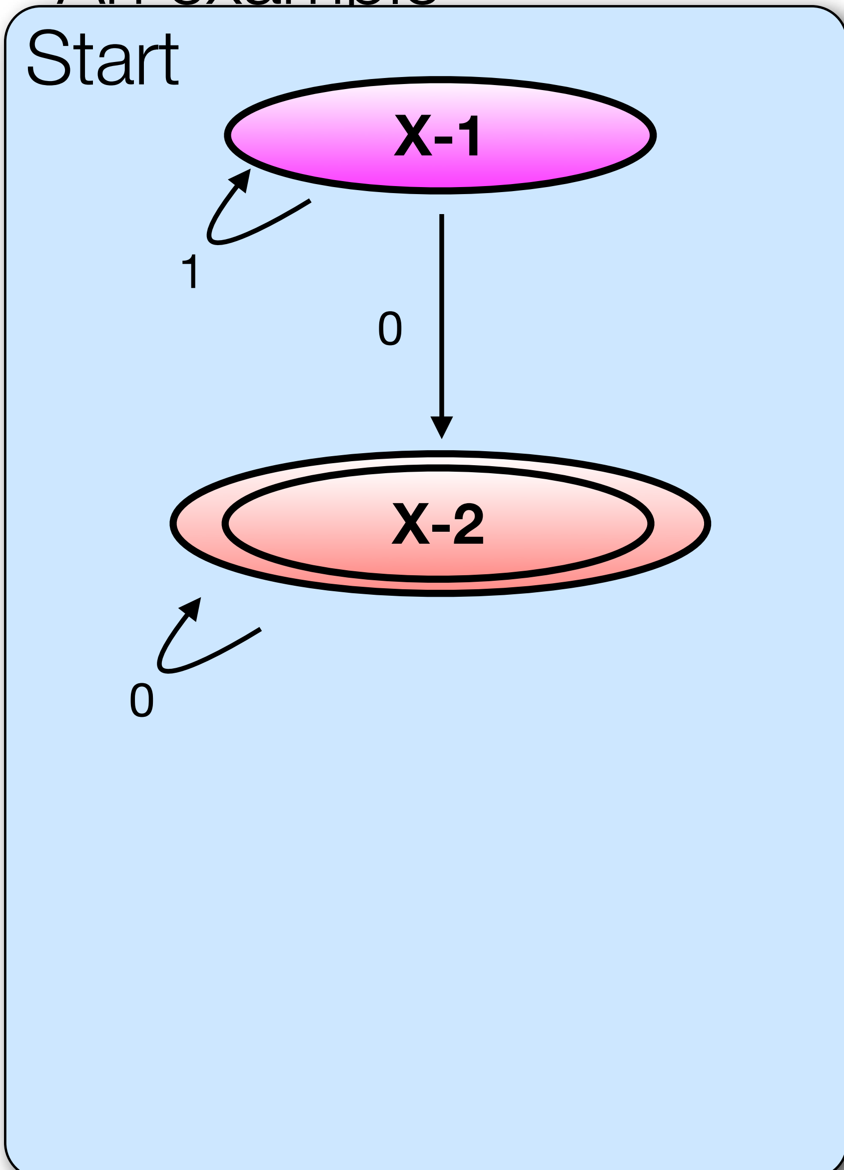
X-1 X-2



State	0	1
SABF	X-2	X-1
BDF	X-2	X-1
CGEF	X-2	N/A
GEF	X-2	N/A

An example

X-1 X-2



State	0	1
SABF	X-2	X-1
BDF	X-2	X-1
CGEF	X-2	N/A
GEF	X-2	N/A

Scanner Code

- Can create Scanner from the DFA one of two ways:
  - Nested case statements (Handwritten)
  - Tables (easy to generate from code, hard to write by hand)

```
state := start
loop
    case state of
        start :
            erase text of current token
            case input_char of
            ' ', '\t', '\n', '\r' : no_op
            '[' : state := got_lbrac
            ']' : state := got_rbrac
            ',' : state := got_comma
            . . .
            '(' : state := saw_lparen
            '.' : state := saw_dot
            '<' : state := saw_lthan
            . . .
            'a'..'z', 'A'..'Z' :
                state := in_ident
            '0'..'9' : state := in_int
            . . .
            else error
        . . .
        saw_lparen: case input_char of
            '*' : state := in_comment
            else return lparen
        in_comment: case input_char of
            '*' : state := leaving_comment
            else no_op
        leaving_comment: case input_char of
            ')' : state := start
            else state := in_comment
        . . .
        saw_dot : case input_char of
            '.' : state := got_dotdot
            else return dot
        . . .
        saw_lthan : case input_char of
            '=' : state := got_le
            else return lt
        . . .
        in_ident : case input_char of
            'a'..'z', 'A'..'Z', '0'..'9', '_' : no_op
            else
                look up accumulated token
                    in keyword table
                if found, return keyword
                else return identifier
        . . .
        in_int : case input_char of
            '0'..'9' : no_op
            '.' :
                peek at character beyond input_char;
                    if '0'..'9', state := saw_real_dot
                    else
                        unread peeked-at character
                        return intconst
            'a'..'z', 'A'..'Z', '_' : error
            else return intconst
        . . .
        saw_real_dot : . . .
        . . .
        got_lbrac : return lbrac
        got_rbrac : return rbrac
        got_comma : return comma
        got_dotdot : return dotdot
        got_le : return le
        . . .
    append input_char to text of current token
    read new input_char
```

Two complications--Nested Case

- **Keywords**
  - It is possible to maintain a DFA for keywords, but the number of states would be even larger! So, they are handled as exceptions to the rule.
- “**Dot-Dot**”
  - Pascal uses “..” to denote a range of numbers; however, to determine the meaning of the “..” we need to “look ahead” after reading the first “.” to determine if “.” denotes the end of a token or a beginning of a new token.
    - “3.14” one token
    - “2 .. 5” three tokens

```
state = 1..number of states
action_rec = record
     action : (move, recognize, error)
     new_state : state
```

This code specifies a two-dimensional transition table, which tells “whether to move, return token, or announce error”

```
image := null
repeat
     loop
          read cur_char
          case scan_tab[cur_char, cur_state].action
               move:
                    cur_state := scan_tab[cur_char, cur_state].new_state
               recognize:
                    tok := scan_tab[cur_char, cur_state].token_found
                    exit inner loop
               error:
                    -- print error message and recover; probably start over
          append cur_char to image
     -- end inner loop
until tok not in [white_space, comment]
look image up in keyword_tab and replace tok with appropriate keyword if found
return (tok, image)
```

```
state = 1..number of states
action_rec = record
    action : (move, recognize, error)
    new_state : state
```

A second table tells when we might have hit the end of a token (for backing up)

```
image := null
repeat
    loop
        read cur_char
        case scan_tab[cur_char, cur_state].action
            move:
                cur_state := scan_tab[cur_char, cur_state].new_state
            recognize:
                tok := scan_tab[cur_char, cur_state].token_found
                exit inner loop
            error:
                -- print error message and recover; probably start over
        append cur_char to image
    -- end inner loop
until tok not in [white_space, comment]
look image up in keyword_tab and replace tok with appropriate keyword if found
return (tok, image)
```

Pragmas

- **Pragmas** are “comments” that provide direction for the compiler.
  - For example, “Variable x is used a lot, keep it in memory if possible.”
- These are often handled by the parser since this makes the grammar much simpler.