CS 6109 Compiler Design

Dr. Arockia Xavier Annie R
Asst. Professor, DCSE
CEG, Anna University
Chennai -600025
Email: annie@annauniv.edu

Lexical Analysis

Outline

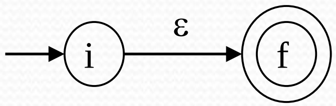
- RE to NFA
- NFA to DFA
- RE to DFA
- Minimizing DFA
- Lexical analyzer generator
- Design of lexical analyzer generator

Converting A Regular Expression into NFA (Thomson's Construction)

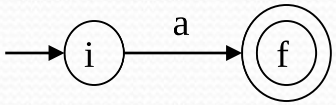
- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
  - It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
  - To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA

Thomson's Construction (cont.)

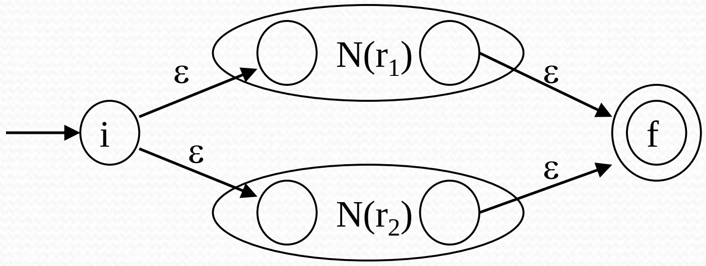
- To recognize an empty string ε



- To recognize a symbol a in the alphabet Σ



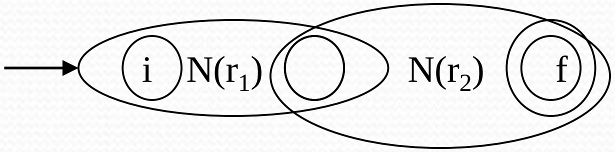
- If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2
  - For regular expression $r_1 \mid r_2$



NFA for $r_1 \mid r_2$

Thomson's Construction (cont.)

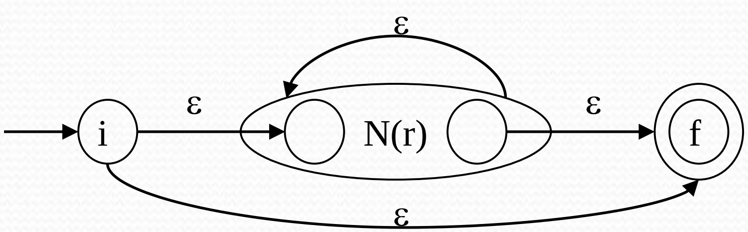
- For regular expression $r_1 \; r_2$



Final state of $N(r_2)$ become final state of $N(r_1r_2)$

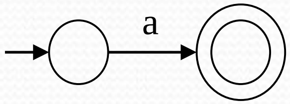
NFA for $r_1 \; r_2$

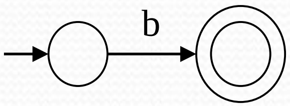
- For regular expression r^*

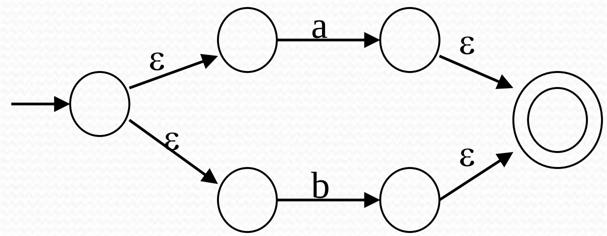


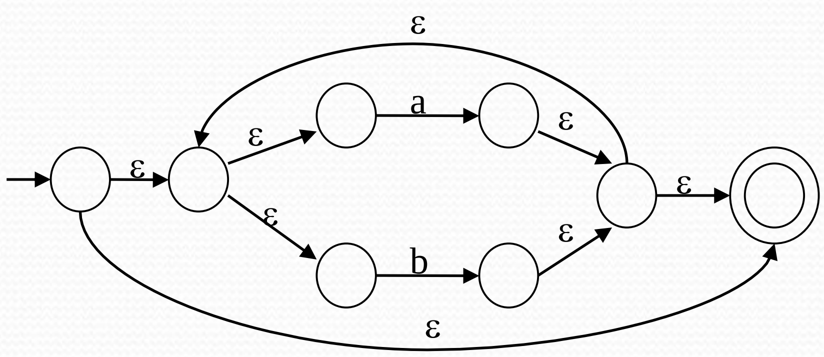
NFA for r^*

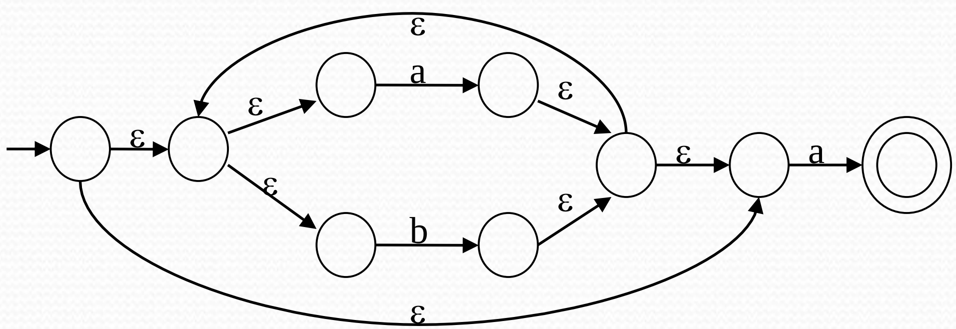
Thomson’s Construction Example - (a|b) * a

a: 

b: 

(a | b) 

(a|b) * 

(a|b) * a 

Converting a NFA into a DFA (subset constr)

put ε-closure($\{s_o\}$) as an unmarked state into the set of DFA (DS)

while (there is one unmarked S_1 in DS) do

begin

mark S_1

ε-closure($\{s_0\}$) is the set of all states can be accessible from s_0 by ε-transition.

for each input symbol a do

set of states to which there is a transition on a from a state s in S_1

begin

$S_2 \leftarrow \varepsilon$-closure(move($S_1$,a))

if (S_2 is not in DS) then

add S_2 into DS as an unmarked state

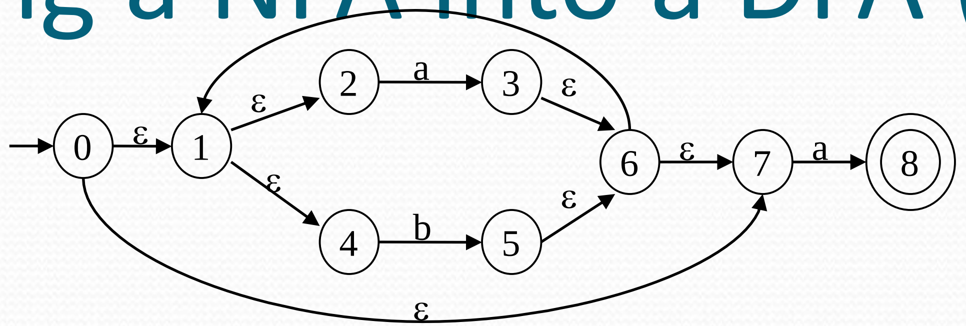
transfunc[S_1,a] $\leftarrow S_2$

end

end

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure($\{s_o\}$)

Converting a NFA into a DFA (Example)



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into DS as an unmarked state

⇓ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into DS

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into DS

transfunc[S_0,a] ← S_1 transfunc[S_0,b] ← S_2

⇓ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_1,a] ← S_1 transfunc[S_1,b] ← S_2

⇓ mark S_2

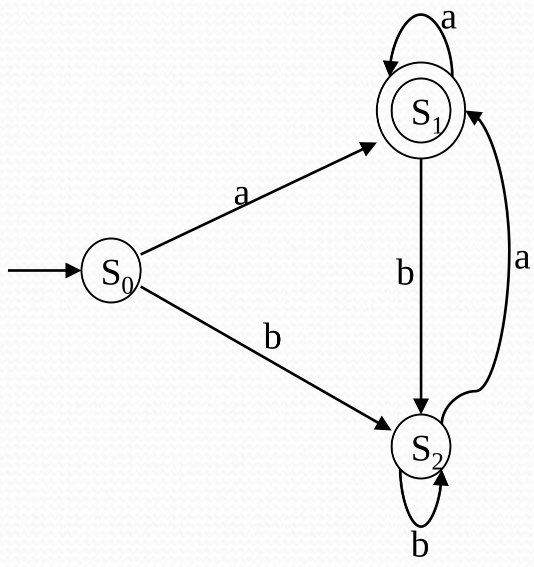
ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_2,a] ← S_1 transfunc[S_2,b] ← S_2

Converting a NFA into a DFA (Example – cont.)

S_0 is the start state of DFA since 0 is a member of $S_0=\{0,1,2,4,7\}$
S_1 is an accepting state of DFA since 8 is a member of $S_1 = \{1,2,3,4,6,7,8\}$



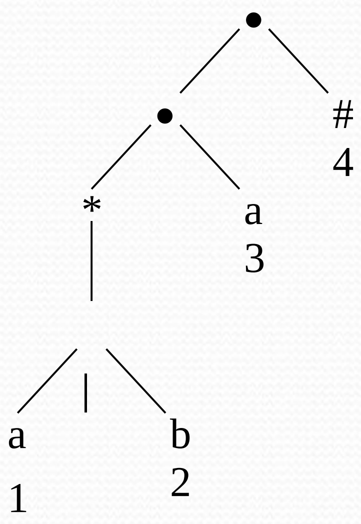
Converting Regular Expressions Directly to DFAs

- We may convert a regular expression into a DFA (without creating a NFA first).
- First we augment the given regular expression by concatenating it with a special symbol #.

  r ➔ (r)# augmented regular expression
- Then, we create a syntax tree for this augmented regular expression.
- In this syntax tree, all alphabet symbols (plus # and the empty string) in the augmented regular expression will be on the leaves, and all inner nodes will be the operators in that augmented regular expression.
- Then each alphabet symbol (plus #) will be numbered (position numbers).

Regular Expression ➔ DFA (cont.)

$(a|b)^* a$ ➔ $(a|b)^* a \#$ augmented regular expression



Syntax tree of $(a|b)^* a \#$

- each symbol is numbered (positions)
- each symbol is at a leave

- inner nodes are operators

followpos

Then we define the function **followpos** for the positions (positions assigned to leaves).

followpos(i) -- is the set of positions which can follow the position 'i' in the strings generated by the augmented regular expression.

For example, $(a \mid b)^* \; a \;\; \#$

$\quad\;\; 1 \;\; 2 \quad\;\; 3 \;\; 4$

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

followpos is just defined for leaves, it is not defined for inner nodes.

firstpos, lastpos, nullable

- To evaluate followpos, we need three more functions to be defined for the nodes (not just for leaves) of the syntax tree.
- **firstpos(n)** -- the set of the positions of the **first** symbols of strings generated by the sub-expression rooted by n.
- **lastpos(n)** -- the set of the positions of the **last** symbols of strings generated by the sub-expression rooted by n.
- **nullable(n)** -- *true* if the empty string is a member of strings generated by the sub expression rooted by n *false* otherwise

How to evaluate firstpos, lastpos, nullable

n	nullable(n)	firstpos(n)	lastpos(n)
leaf labeled ε	true	Φ	Φ
leaf labeled with position i	false	{i}	{i}
$\mid$ c_1 c_2	nullable(c_1) or nullable(c_2)	firstpos(c_1) ∪ firstpos(c_2)	lastpos(c_1) ∪ lastpos(c_2)
$\bullet$ c_1 c_2	nullable(c_1) and nullable(c_2)	if (nullable(c_1)) firstpos(c_1) ∪ firstpos(c_2) else firstpos(c_1)	if (nullable(c_2)) lastpos(c_1) ∪ lastpos(c_2) else lastpos(c_2)
* c_1	true	firstpos(c_1)	lastpos(c_1)

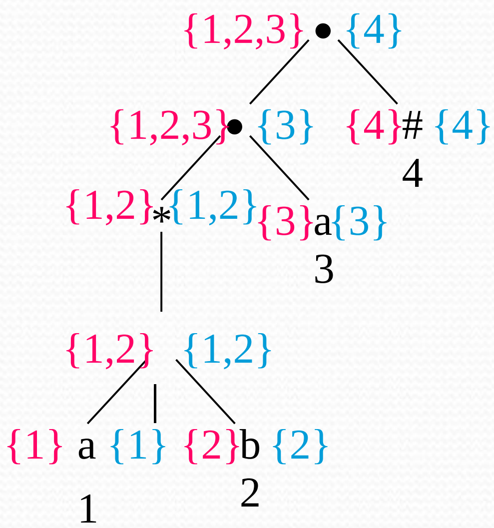
How to evaluate followpos

- Two-rules define the function followpos:

1. If **n** is concatenation-node with left child c_1 and right child c_2, and **i** is a position in **lastpos**($\mathbf{c_1}$), then all positions in **firstpos**($\mathbf{c_2}$) are in **followpos**(**i**).
2. If **n** is a star-node, and **i** is a position in **lastpos**(**n**), then all positions in **firstpos**(**n**) are in **followpos**(**i**).

- If firstpos and lastpos have been computed for each node, followpos of each position can be computed by making one depth-first traversal of the syntax tree.

Example -- (a | b)* a



pink – firstpos
blue – lastpos

Then we can calculate followpos

followpos(1) = {1,2,3}
followpos(2) = {1,2,3}
followpos(3) = {4}
followpos(4) = {}

- After we calculate follow positions, we are ready to create DFA for the regular expression.

Algorithm (RE ➔ DFA)

- Create the syntax tree of (r) #
- Calculate the functions: followpos, firstpos, lastpos, nullable
- Put firstpos(root) into the states of DFA as an unmarked state.
- *while* (there is an unmarked state S in the states of DFA) *do*
  - mark **S**
  - *for each* input symbol **a** *do*
    - let $s_1,\ldots,s_n$ are positions in **S** and symbols in those positions are **a**
    - **S'** 🡐 followpos(s_1) $\cup \ldots \cup$ followpos(s_n)
    - **move(S,a)** 🡐 **S'**
    - if (**S'** is not empty and not in the states of DFA)
      - put **S'** into the states of DFA as an unmarked state.
- *the start state of DFA is firstpos(root)*
- *the accepting states of DFA are all states containing the position of #*

Example -- $(a \mid b)^* a \#$

1 2 3 4

followpos(1)={1,2,3}   followpos(2)={1,2,3}   followpos(3)={4}
followpos(4)={}

S_1=firstpos(root)={1,2,3}

$\Downarrow$ mark S_1

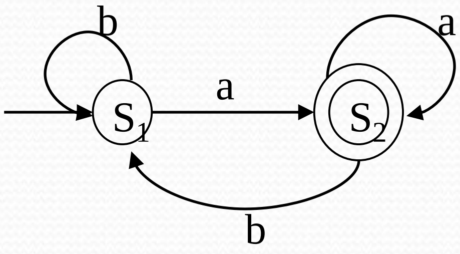
a: followpos(1) $\cup$ followpos(3)={1,2,3,4}=S_2 move(S_1,a)=S_2

b: followpos(2)={1,2,3}=S_1 move(S_1,b)=S_1

$\Downarrow$ mark S_2

a: followpos(1) $\cup$ followpos(3)={1,2,3,4}=S_2 move(S_2,a)=S_2

b: followpos(2)={1,2,3}=S_1 move(S_2,b)=S_1



start state: S_1

accepting states: {S_2}

Example -- (a | ε) b c*

$$\begin{array}{cccccc} (& \underset{1}{a} & \mid \ \varepsilon) & \underset{2}{b} & \underset{3}{c^*} & \underset{4}{\#} \end{array}$$

followpos(1)={2}   followpos(2)={3,4}   followpos(3)={3,4}
followpos(4)={}

S_1=firstpos(root)={1,2}

⇓ mark S_1

a: followpos(1)={2}=S_2 move(S_1,a)=S_2

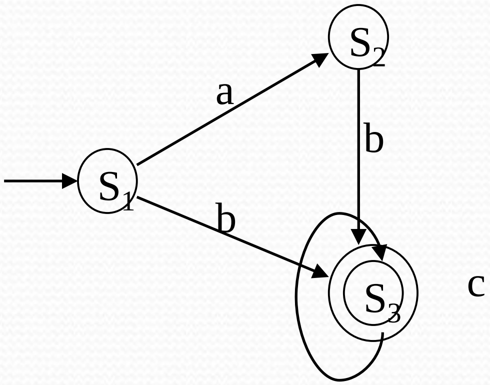
b: followpos(2)={3,4}=S_3 move(S_1,b)=S_3

⇓ mark S_2

b: followpos(2)={3,4}=S_3 move(S_2,b)=S_3

⇓ mark S_3

c: followpos(3)={3,4}=S_3 move(S_3,c)=S_3



start state: S_1

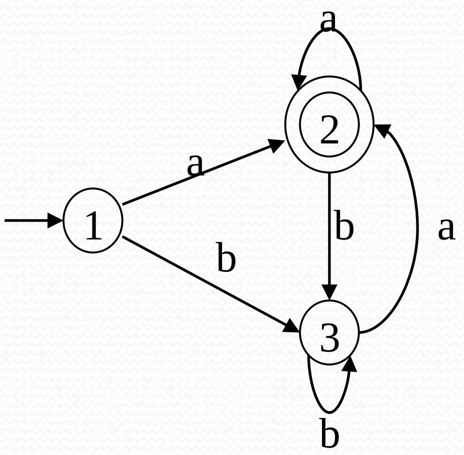
accepting states: {S_3}

Minimizing Number of States of a DFA

- partition the set of states into two groups:
  - G_1 : set of accepting states
  - G_2 : set of non-accepting states
- For each new group G
  - partition G into subgroups such that states s_1 and s_2 are in the same group iff

    for all input symbols a, states s_1 and s_2 have transitions to states in the same group.
- Start state of the minimized DFA is the group containing the start state of the original DFA.
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

Minimizing DFA - Example

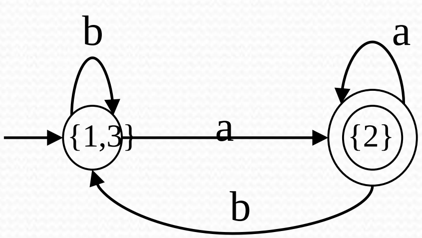


$G_1 = \{2\}$
$G_2 = \{1,3\}$

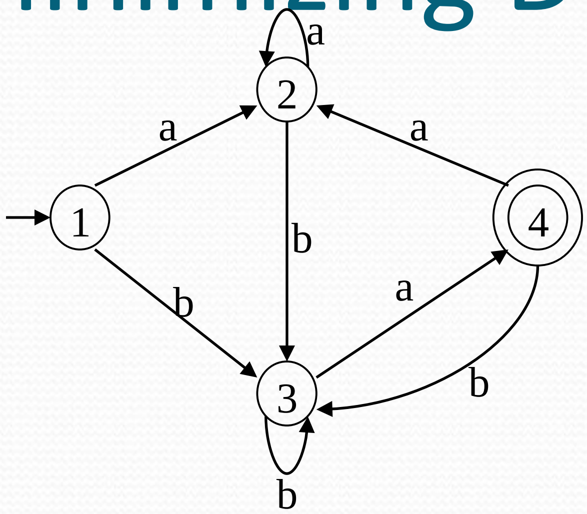
G_2 cannot be partitioned because

move(1,a)=2 move(1,b)=3
move(3,a)=2 move(2,b)=3

So, the minimized DFA (with minimum states)



Minimizing DFA – Another Example



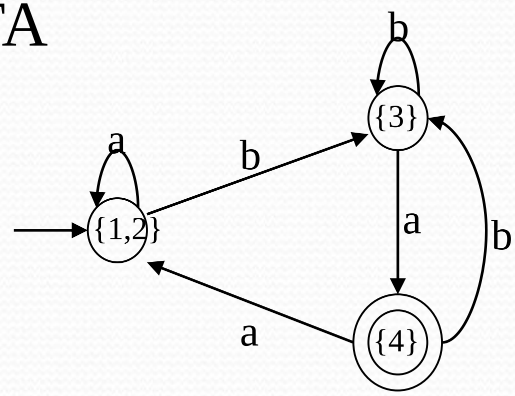
Groups: {1,2,3} {4}

{1,2} {3}

no more partitioning

a	b
1->2	1->3
2->2	2->3
3->4	3->3

So, the minimized DFA



Some Other Issues in Lexical Analyzer

- The lexical analyzer has to recognize the longest possible string.
  - Ex: identifier `newval` -- `n` `ne` `new` `newv` `newva` `newval`
- What is the end of a token? Is there any character which marks the end of a token?
  - It is normally not defined.
  - If the number of characters in a token is fixed, in that case no problem: + -
  - But < ➔ < or <> (in Pascal)
  - The end of an identifier : the characters cannot be in an identifier can mark the end of token.
  - We may need a lookhead
  - In Prolog: p :- X is 1. p :- X is 1.5.

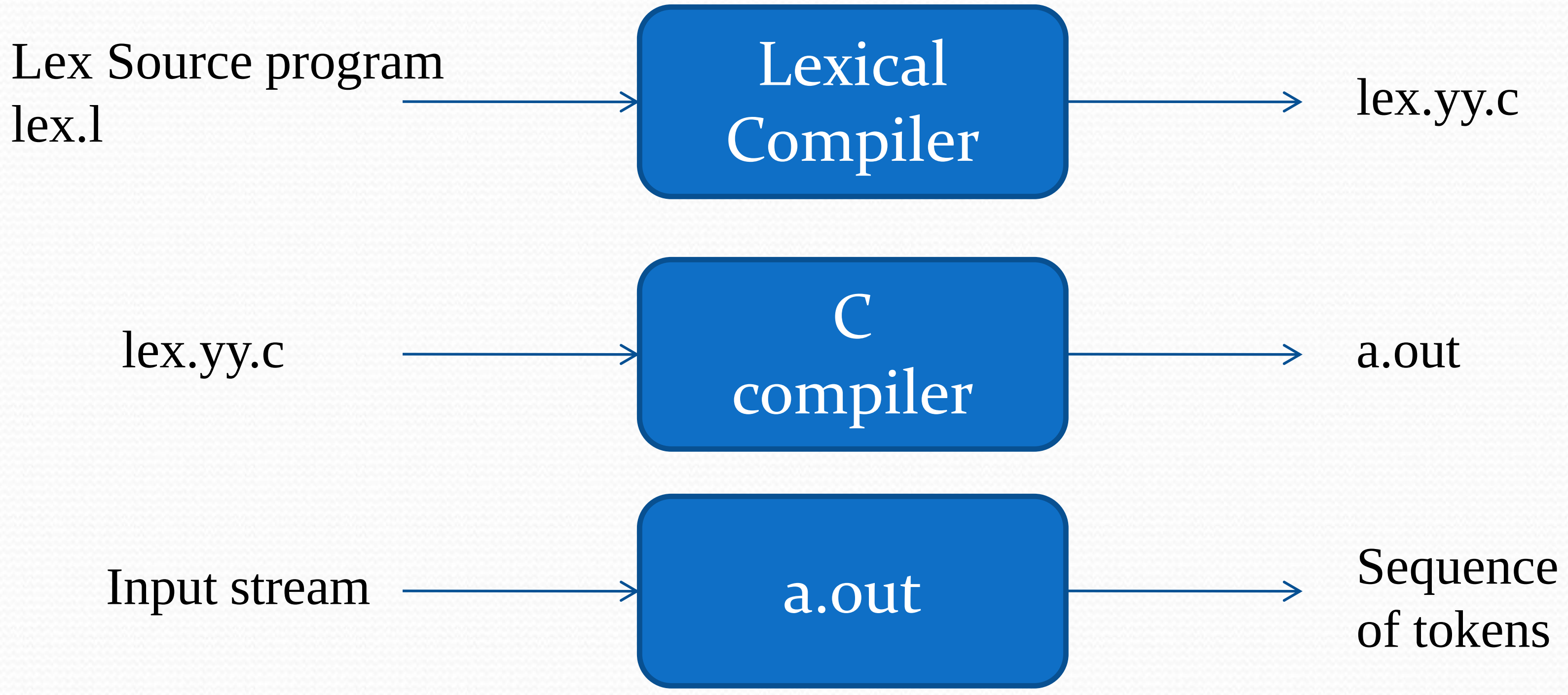
Some Issues in Lexical Analyzer (cont.)

- Skipping comments
  - Normally we don't return a comment as a token.
  - We skip a comment, and return the next token (which is not a comment) to the parser.
  - So, the comments are only processed by the lexical analyzer, and the don't complicate the syntax of the language.
- Symbol table interface
  - symbol table holds information about tokens (at least lexeme of identifiers)
  - how to implement the symbol table, and what kind of operations.
    - hash table – open addressing, chaining
    - putting into the hash table, finding the position of a token from its lexeme.
- Positions of the tokens in the file (for the error handling).

Architecture of a transition-diagram-based lexical analyzer

```
TOKEN getRelop()
{
    TOKEN retToken = new (RELOP)
    while (1) {             /* repeat character processing until a
                                   return or failure occurs */
    switch(state) {
         case 0: c= nextchar();
                   if (c == '<') state = 1;
                   else if (c == '=') state = 5;
                   else if (c == '>') state = 6;
                   else fail();  /* lexeme is not a relop */
                   break;
         case 1: …
         …
         case 8: retract();
                  retToken.attribute = GT;
                  return(retToken);
    }
```

Lexical Analyzer Generator - Lex



Structure of Lex programs

declarations
%%
translation rules → Pattern {Action}
%%
auxiliary functions

Example

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID); }
{number}    {yylval = (int) installNum(); return(NUMBER);}
…
```

```
Int installID() {/* funtion to install the
    lexeme, whose first character is
    pointed to by yytext, and whose
    length is yyleng, into the symbol
    table and return a pointer thereto
    */
}

Int installNum() { /* similar to
    installID, but puts numerical
    constants into a separate table */
}
```

Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\rightarrow^{\text{input}}$ state

Finite Automata

- Transition

$$s_1 \rightarrow^a s_2$$

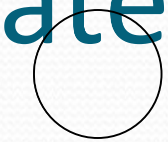
- Is read

In state s_1 on input "a" go to state s_2

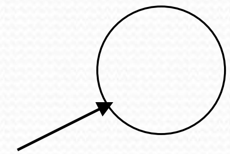
- If end of input
  - If in accepting state => accept, othewise => reject
- If no transition possible => reject

Finite Automata State Graphs

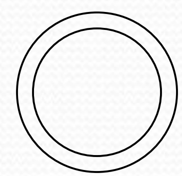
- A state



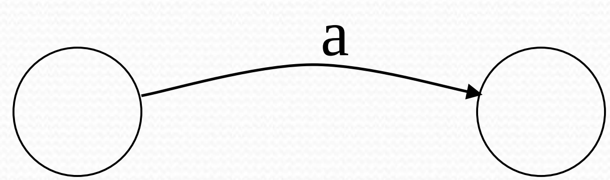
- The start state



- An accepting state

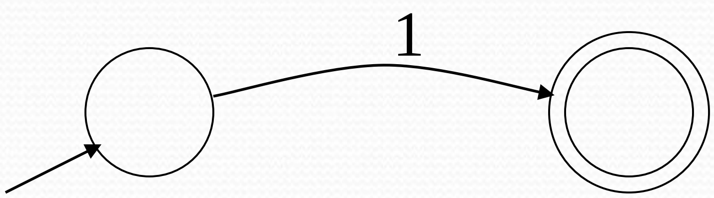


- A transition



A Simple Example

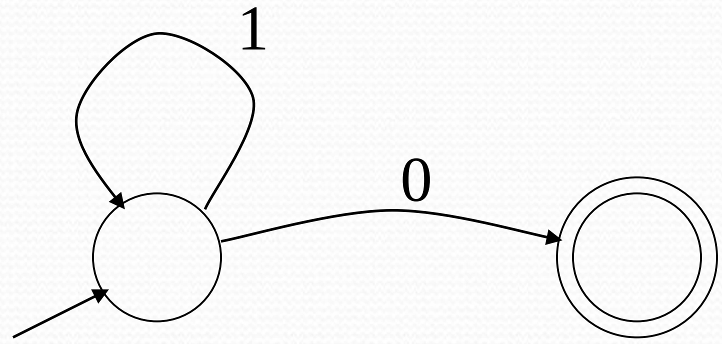
- A finite automaton that accepts only “1”



- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

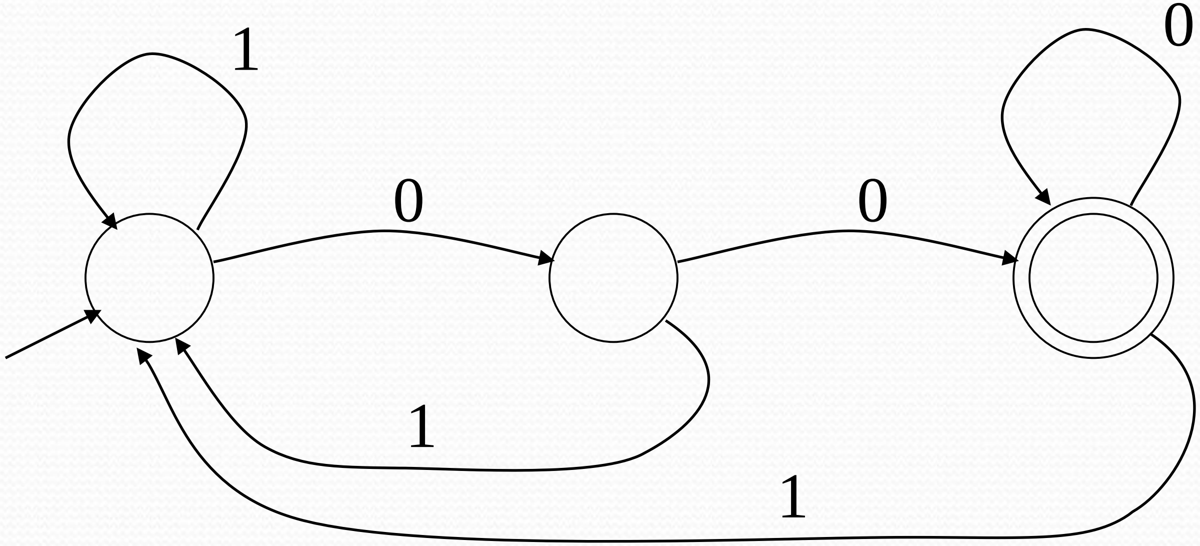
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



- Check that "1110" is accepted but "110…" is not

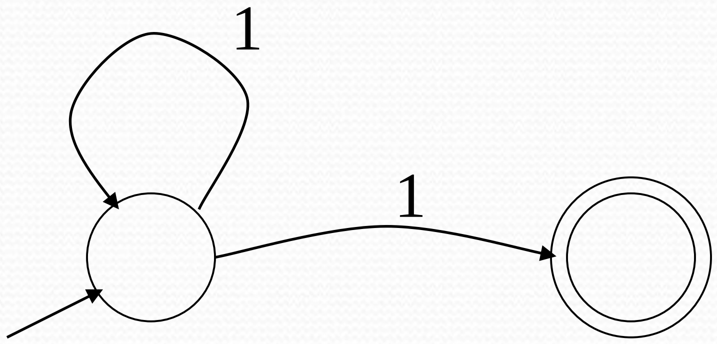
And Another Example

- Alphabet {0,1}
- What language does this recognize?



And Another Example

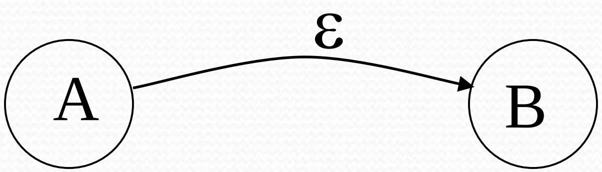
- Alphabet still { 0, 1 }



- The operation of the automaton is not completely defined by the input
  - On input "11" the automaton could be in either state

Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

Deterministic and Nondeterministic Automata

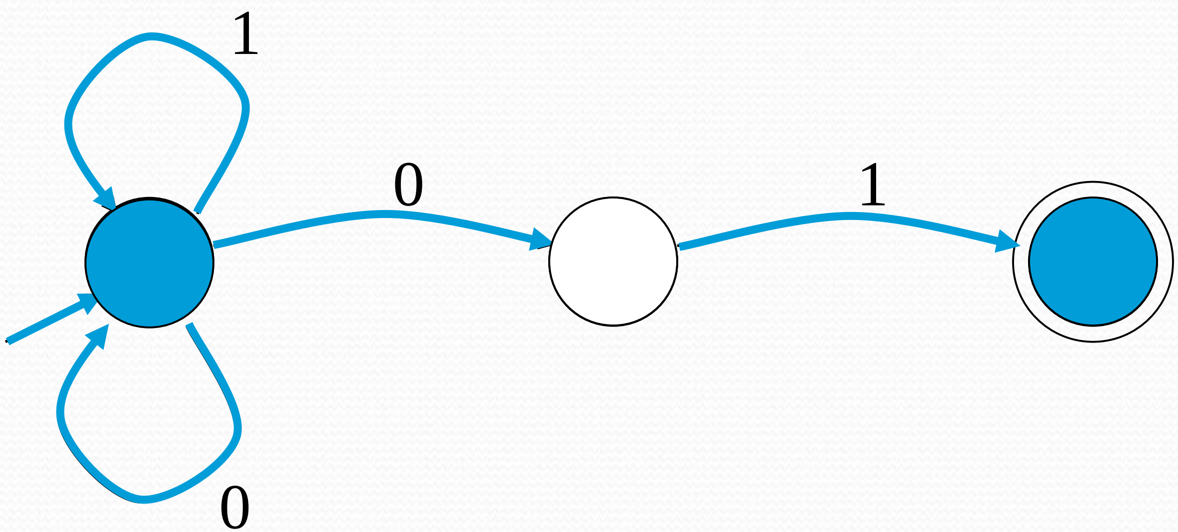
- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it <u>can</u> get in a final state

NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)

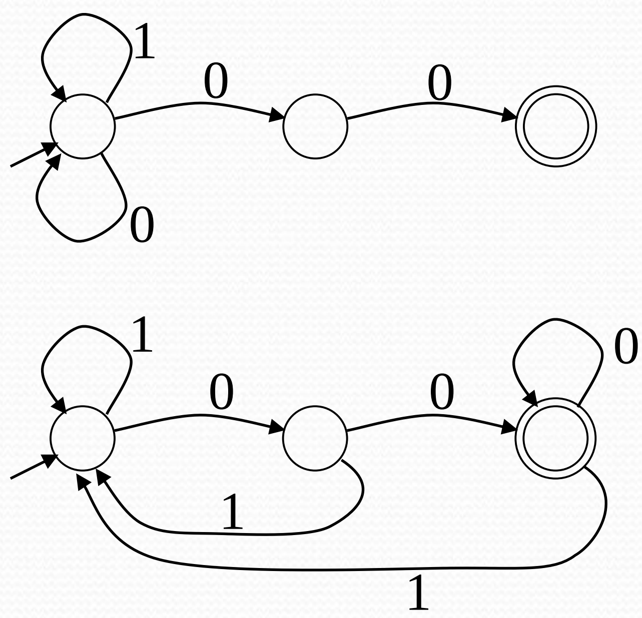
- DFAs are easier to implement
  - There are no choices to consider

NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA

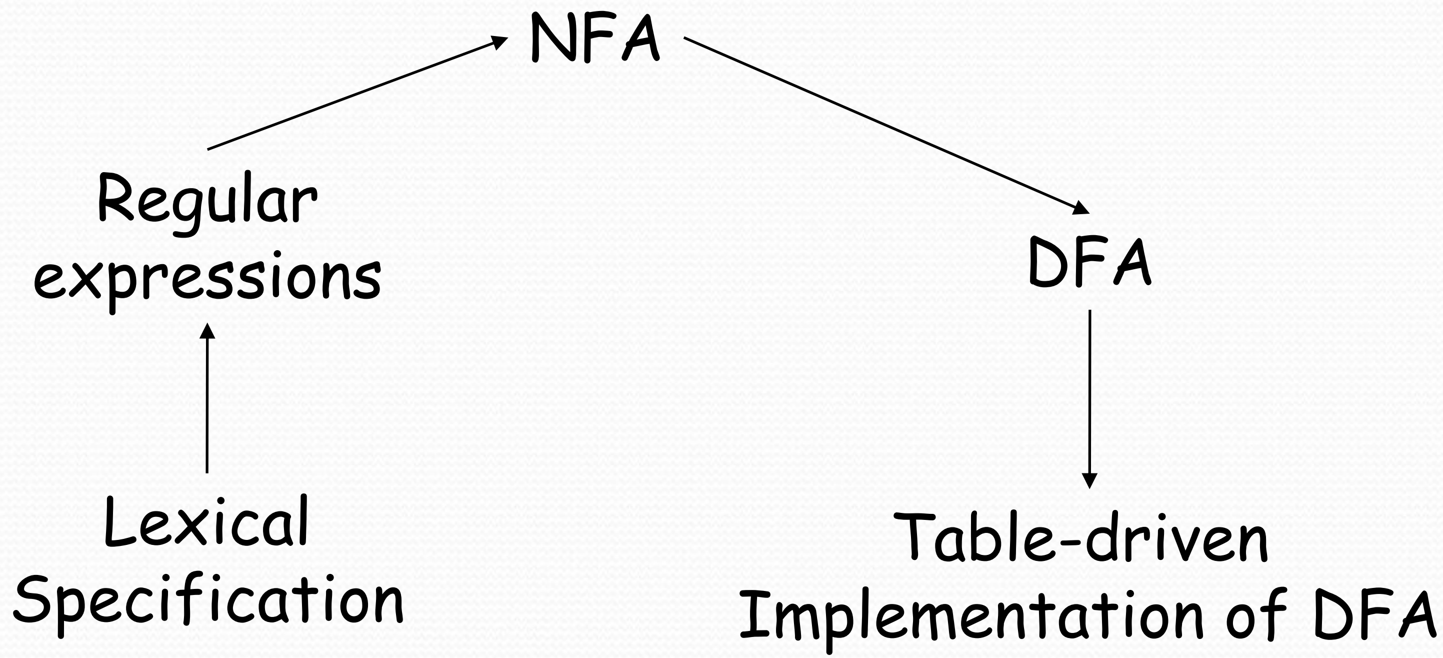
DFA



- DFA can be exponentially larger than NFA

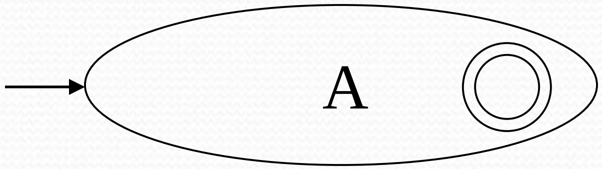
Regular Expressions to Finite Automata

- High-level sketch

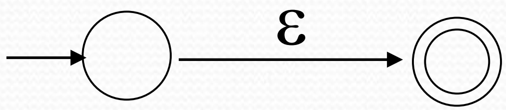


Regular Expressions to NFA (1)

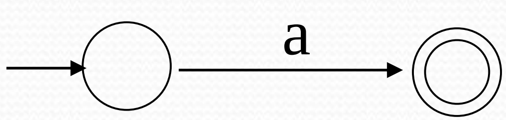
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



- For ε

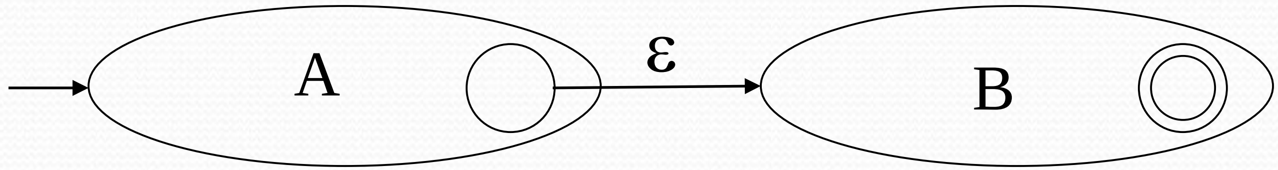


- For input a

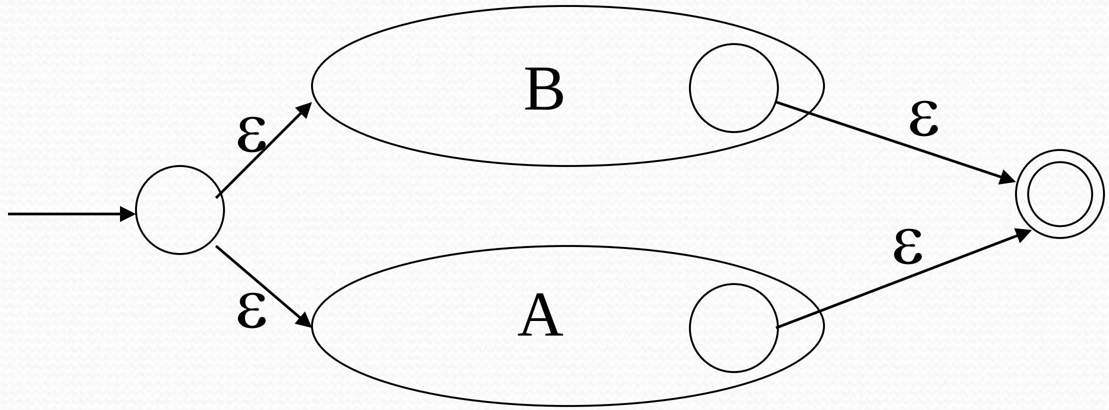


Regular Expressions to NFA (2)

- For AB

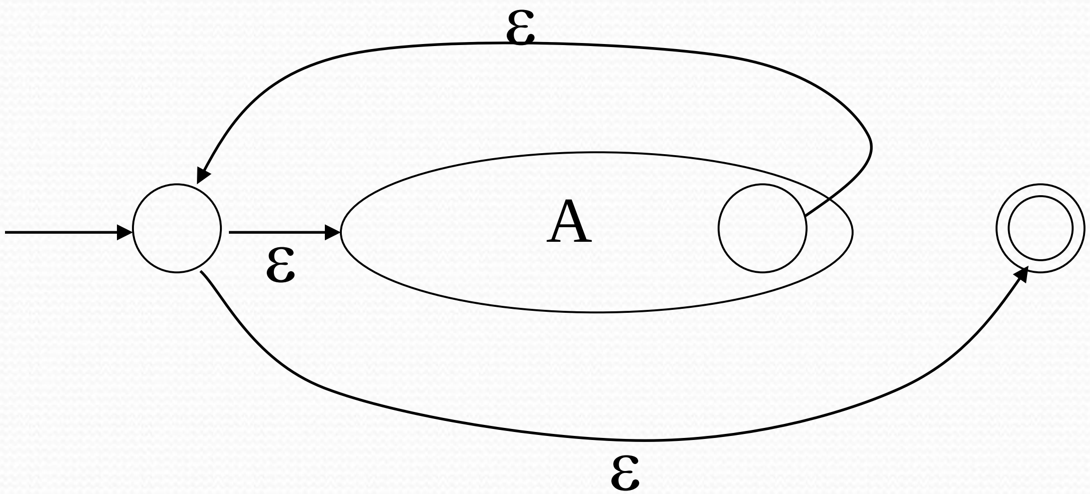


- For A | B



Regular Expressions to NFA (3)

- For A*

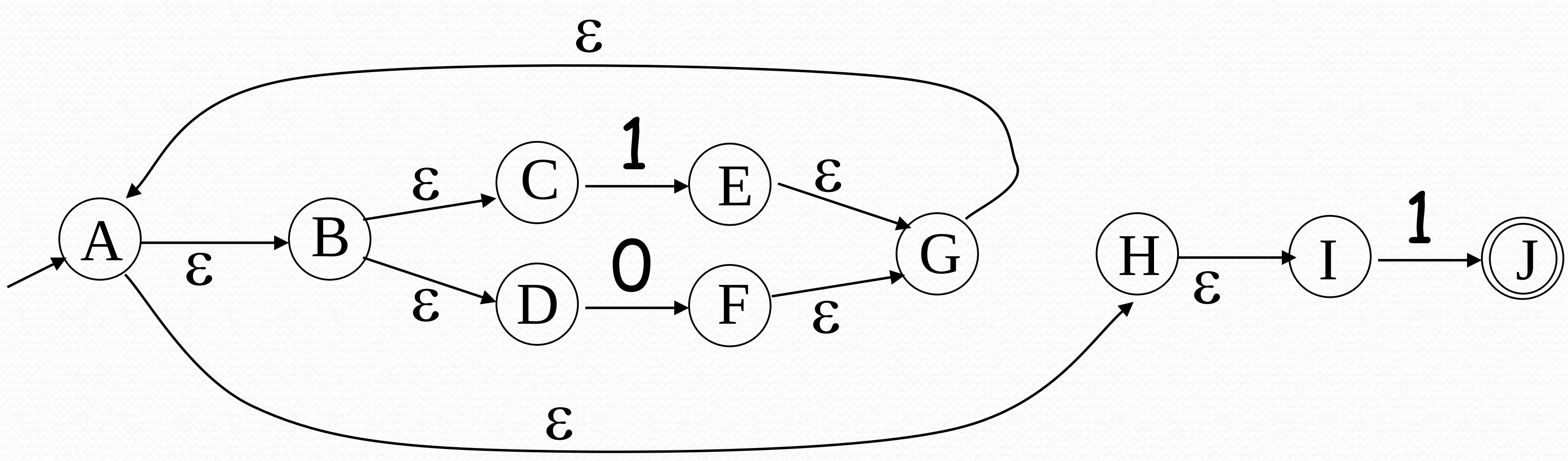


Example of RegExp -> NFA conversion

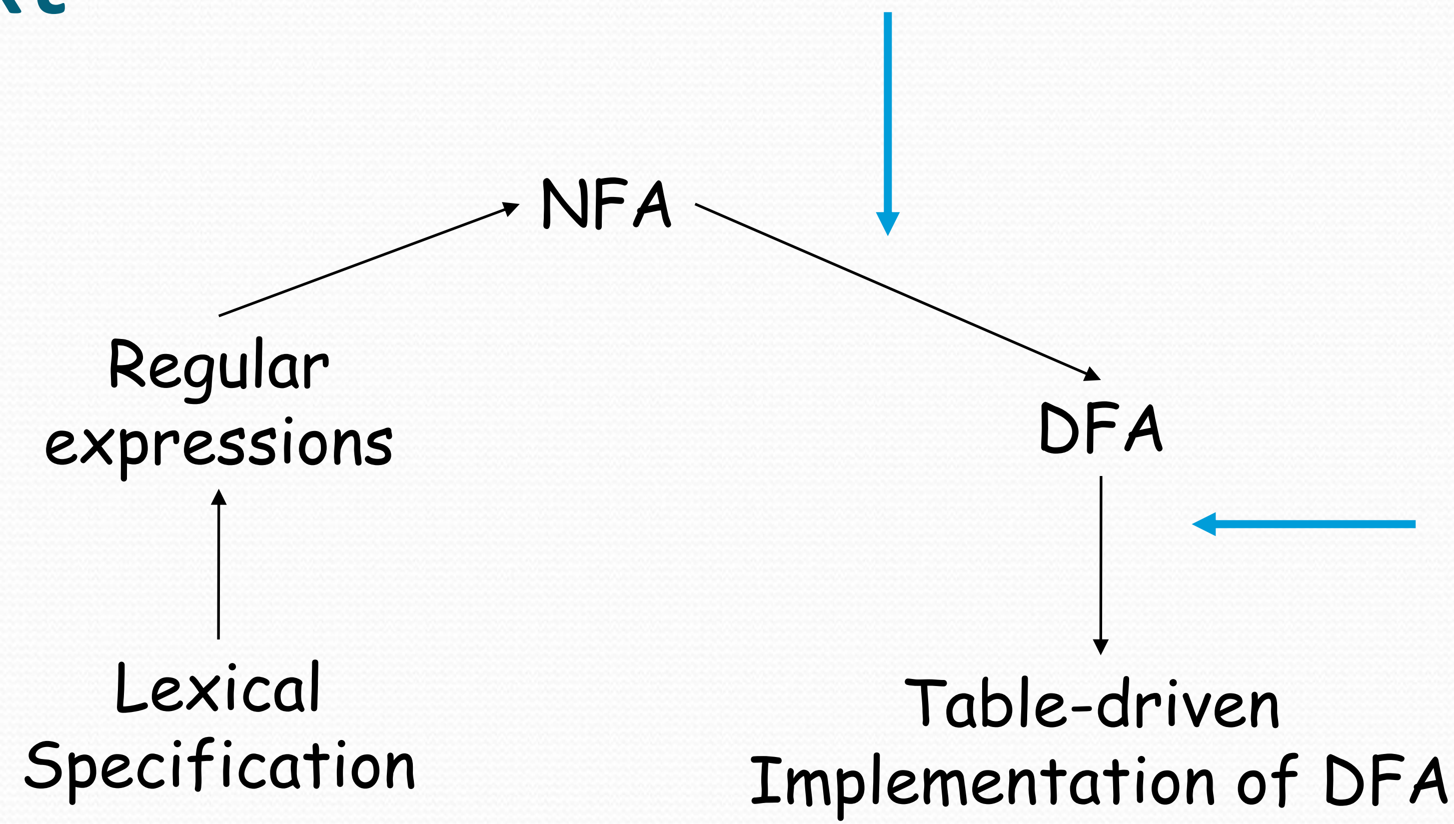
- Consider the regular expression

$$(1 \mid 0)^*1$$

- The NFA is



Next


NFA
Regular expressions
DFA
Lexical Specification
Table-driven Implementation of DFA

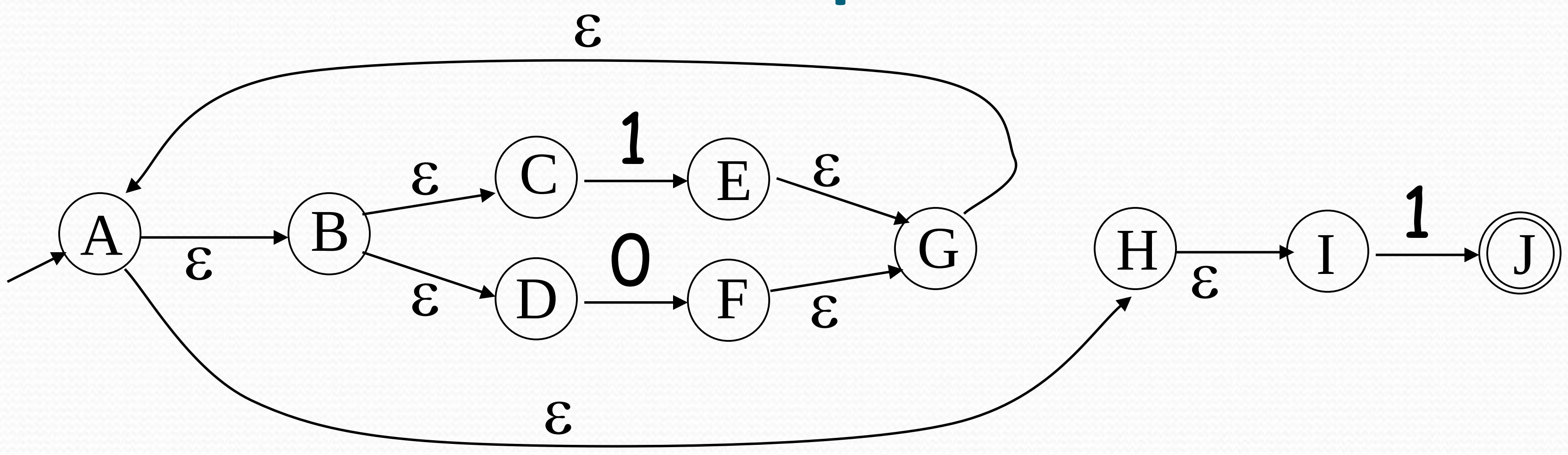
NFA to DFA. The Trick

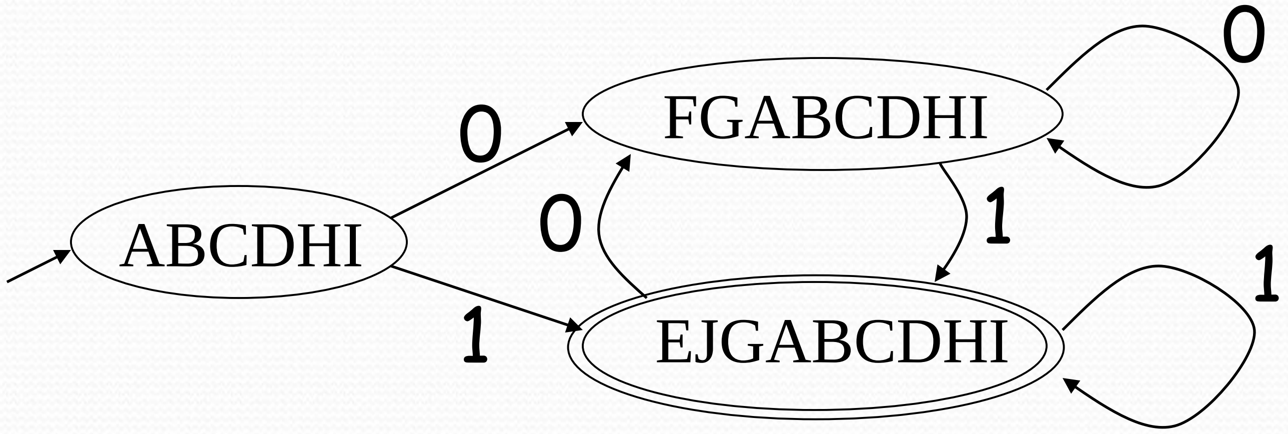
- Simulate the NFA
- Each state of resulting DFA

  = a non-empty subset of states of the NFA
- Start state

  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well

NFA -> DFA Example





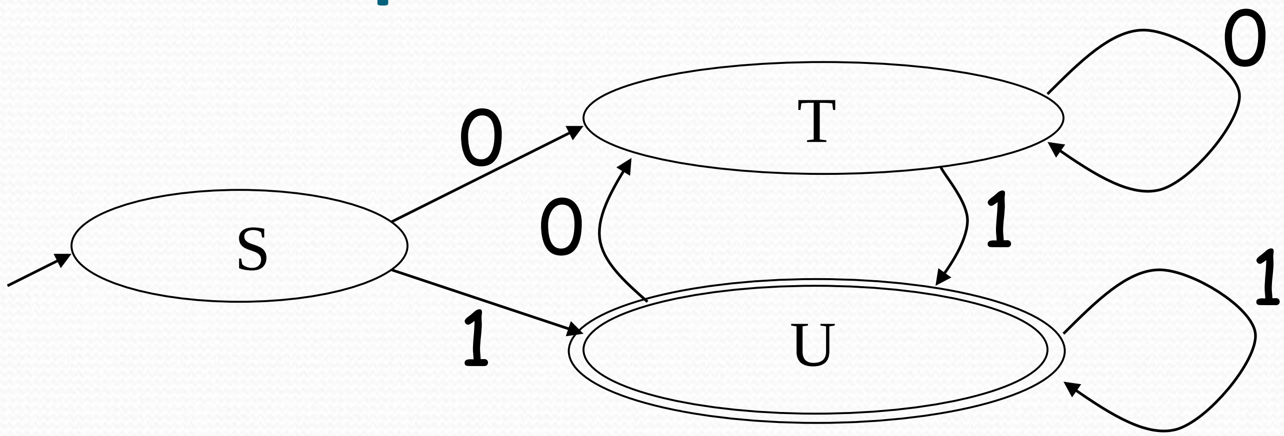
NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
  - $2^N - 1$ = finitely many, but exponentially many

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \rightarrow^a S_k$ define $T[i,a] = k$
- DFA "execution"
  - If in state S_i and input a, read $T[i,a] = k$ and skip to state S_k
  - Very efficient

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex or jflex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

Lexical errors

- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) …
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

Input buffering

- Sometimes lexical analyzer needs to look ahead some symbols to decide about the token to return
  - In C language: we need to look after -, = or < to decide what token to return
  - In Fortran: DO 5 I = 1.25
- We need to introduce a two buffer scheme to handle large look-aheads safely

										E		=		M	*		C	*	*	2	eof										

Sentinels

											E	=	M	eof	*	C	*	*	2	eof													eof

```
Switch (*forward++) {
    case eof:
            if (forward is at end of first buffer) {
                    reload second buffer;
                    forward = beginning of second buffer;
            }
            else if {forward is at end of second buffer) {
                    reload first buffer;\
                    forward = beginning of first buffer;
            }
            else /* eof within a buffer marks the end of input */
                    terminate lexical analysis;
            break;
    cases for the other characters;
}
```

Why to separate Lexical analysis and parsing

1. Simplicity of design
2. Improving compiler efficiency
3. Enhancing compiler portability