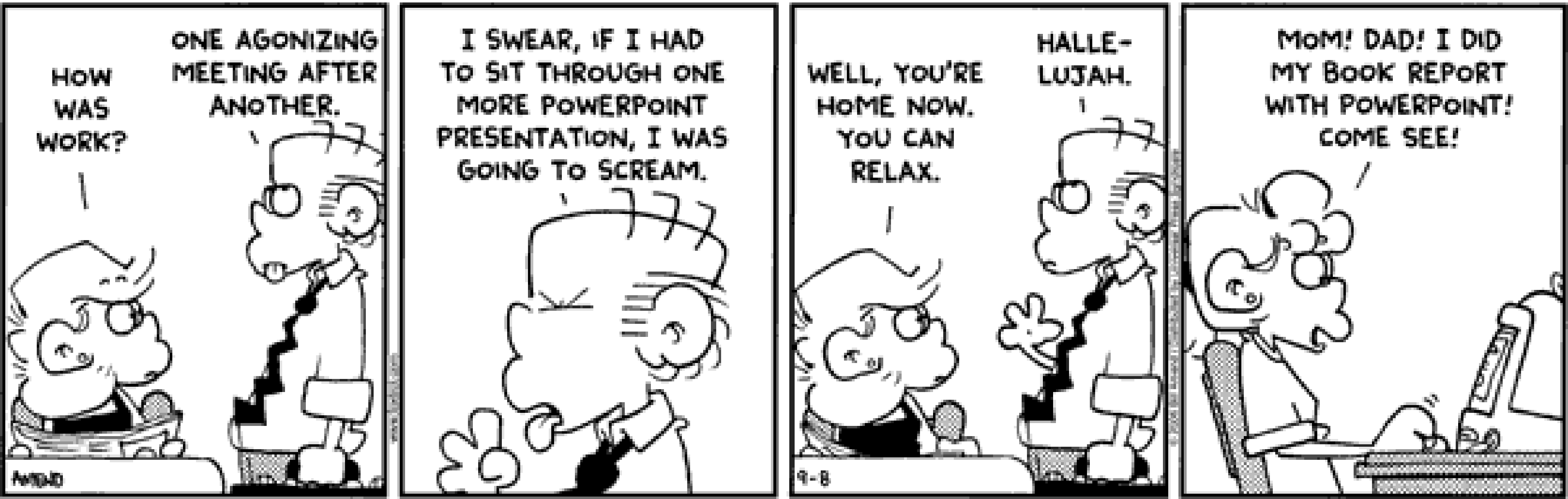
Lexical Analysis

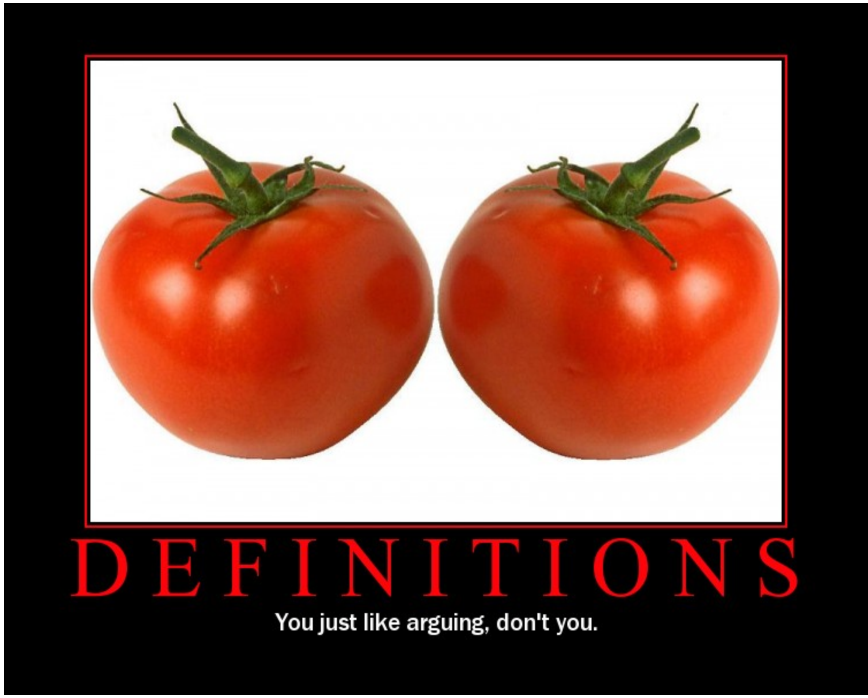
Finite Automata

(Part 2 of 2)



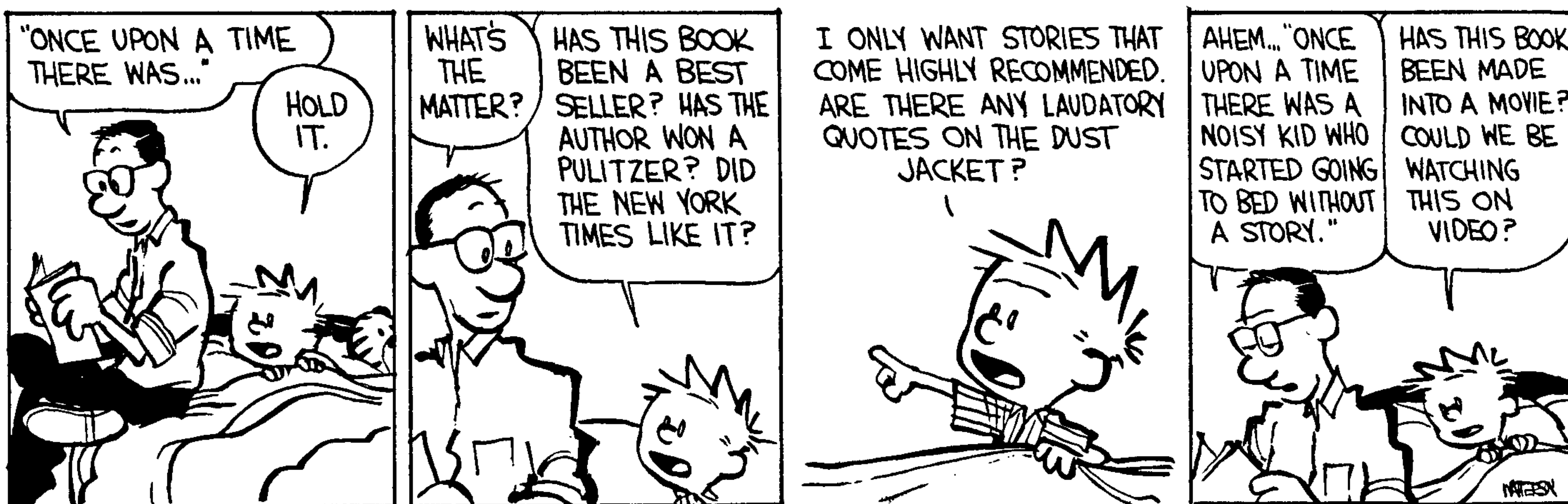
PA0, PA1

- Although we have included the tricky "file ends without a newline" testcases in previous years, students made good cases against them (e.g., they test I/O and not the algorithm) so we are **dropping them from PA1**.
- You can submit new rosetta.yada files for PA1, so you can fix errors from PA0.



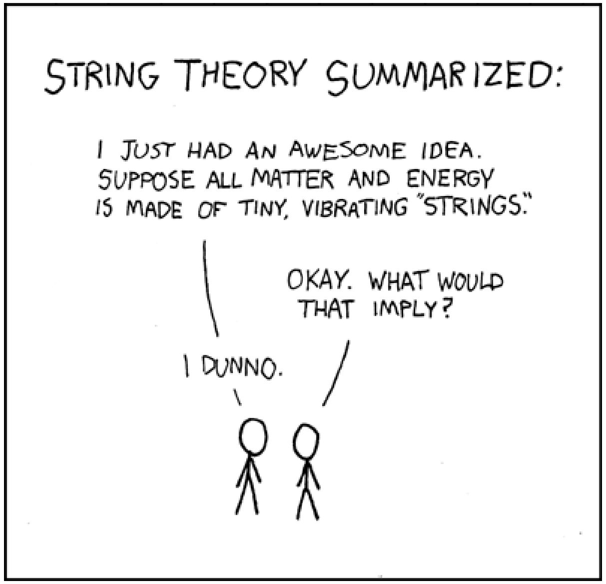
Reading Quiz!

- Are practical parsers and scanners based on deterministic or non-deterministic automata?
- How can regular expressions be used to specify nested constructs?
- How is a two-dimensional *transition table* used in table-driven scanning?



Cunning Plan

- Regular expressions provide a concise notation for **string patterns**
- Use in lexical analysis requires extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known (next)
  - Require only single pass over the input
  - Few operations per character (table lookup)



One-Slide Summary

- **Finite automata** are formal models of computation that can accept regular languages corresponding to regular expressions.
- **Nondeterministic** finite automata (NFA) feature epsilon transitions and multiple outgoing edges for the same input symbol.
- Regular expressions can be **converted** to NFAs.
- Tools will **generate** DFA-based lexer code for you from regular expressions.

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 $\text{state} \rightarrow^{\text{input}} \text{state}$

Finite Automata

- Transition

$$s_1 \rightarrow^{a} s_2$$

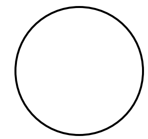
- Is read

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

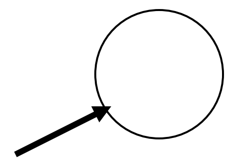
- If end of input (or no transition possible)
  - If in accepting state $\Rightarrow$ accept
  - Otherwise $\Rightarrow$ reject

Finite Automata State Graphs

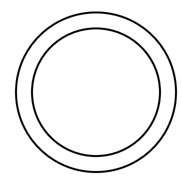
- A state



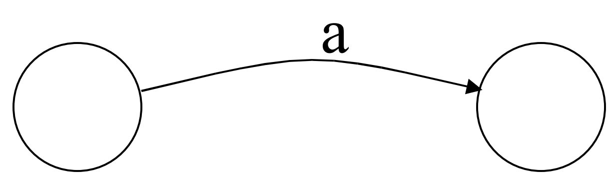
- The start state



- An accepting state

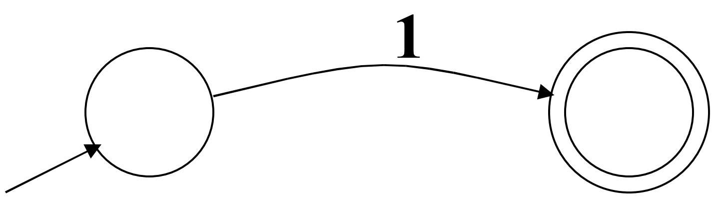


- A transition



A Simple Example

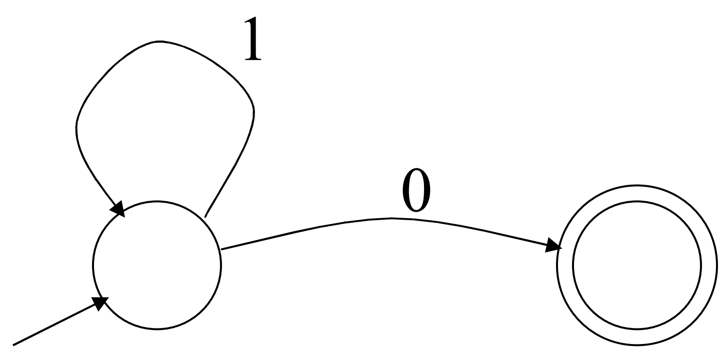
- A finite automaton that accepts only "1"



- 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

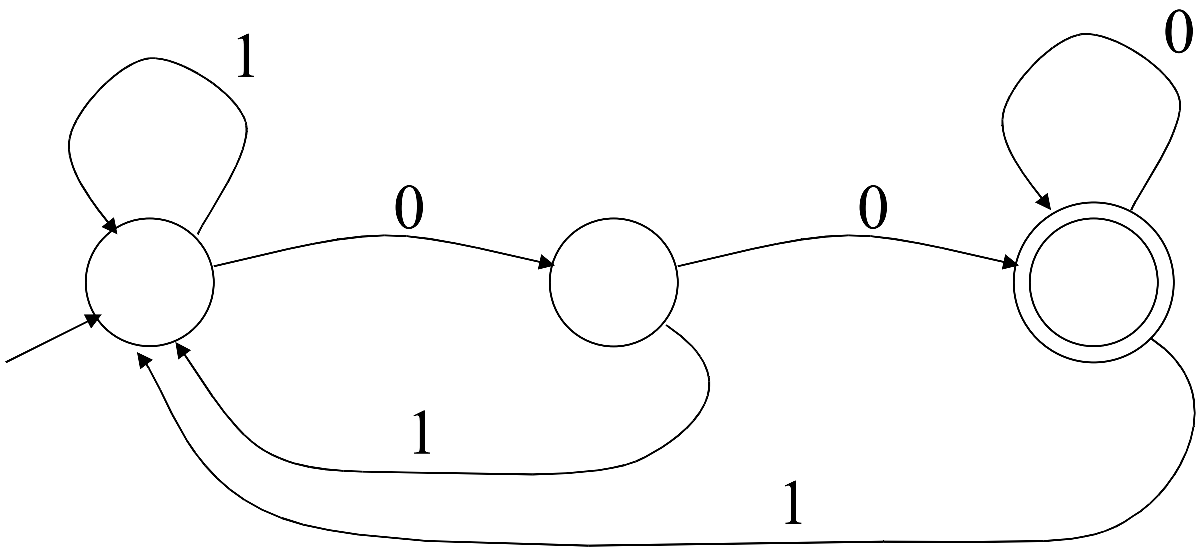
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet $\Sigma = \{0,1\}$

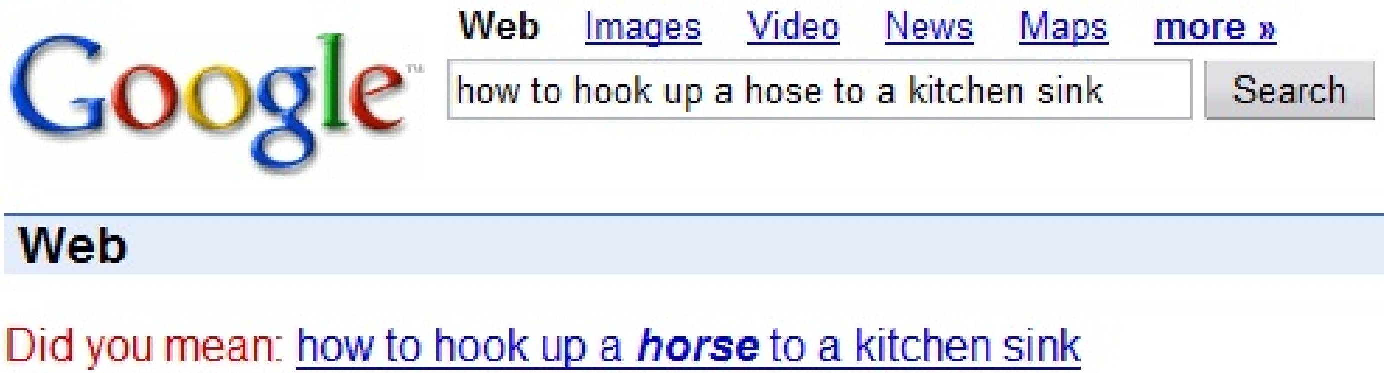


- Check that "**1110**" is accepted but "**110…**" is not

And Another Example

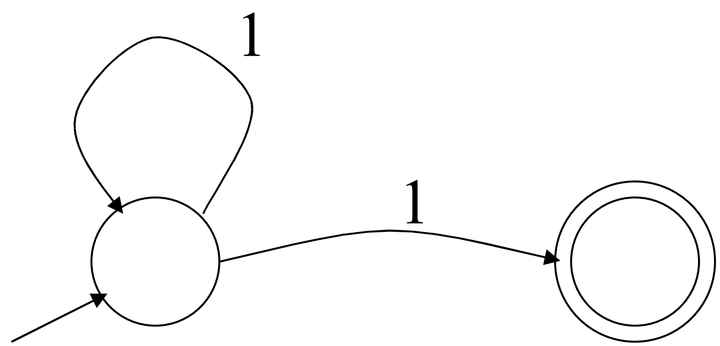
- Alphabet $\Sigma = \{0,1\}$
- What language does this recognize?





And Another Example

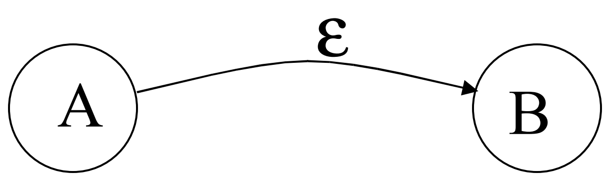
- Alphabet still $\Sigma = \{ 0, 1 \}$



- 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

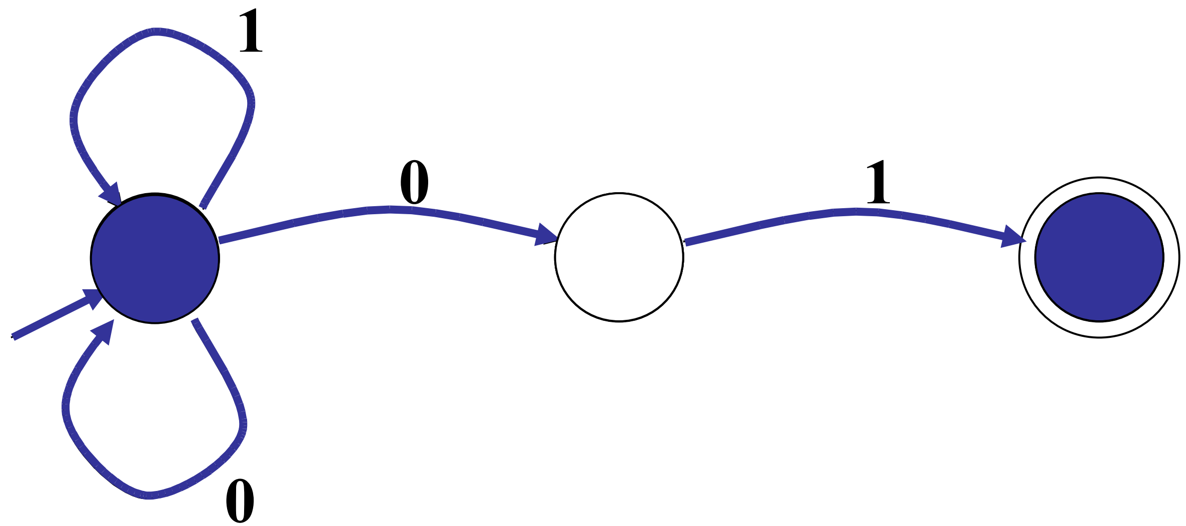
Execution of Finite Automata

- 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

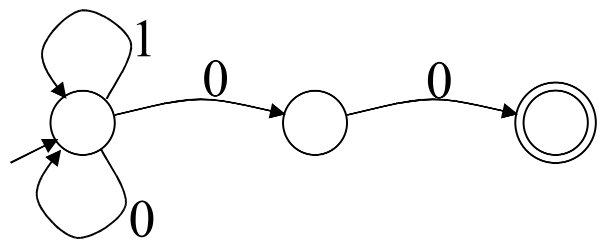
NFA vs. DFA (1)

- NFAs and DFAs recognize the *same* set of languages (regular languages)
  - They have the same expressive power
- DFAs are easier to implement
  - There are no choices to consider

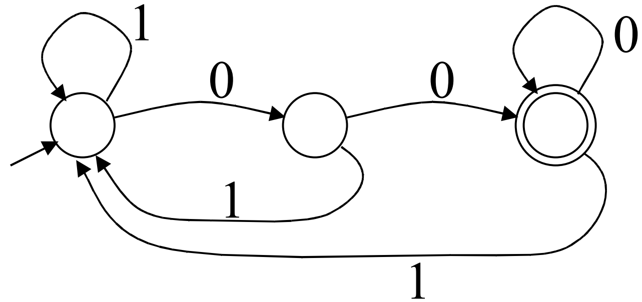
NFA vs. DFA (2)

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

NFA



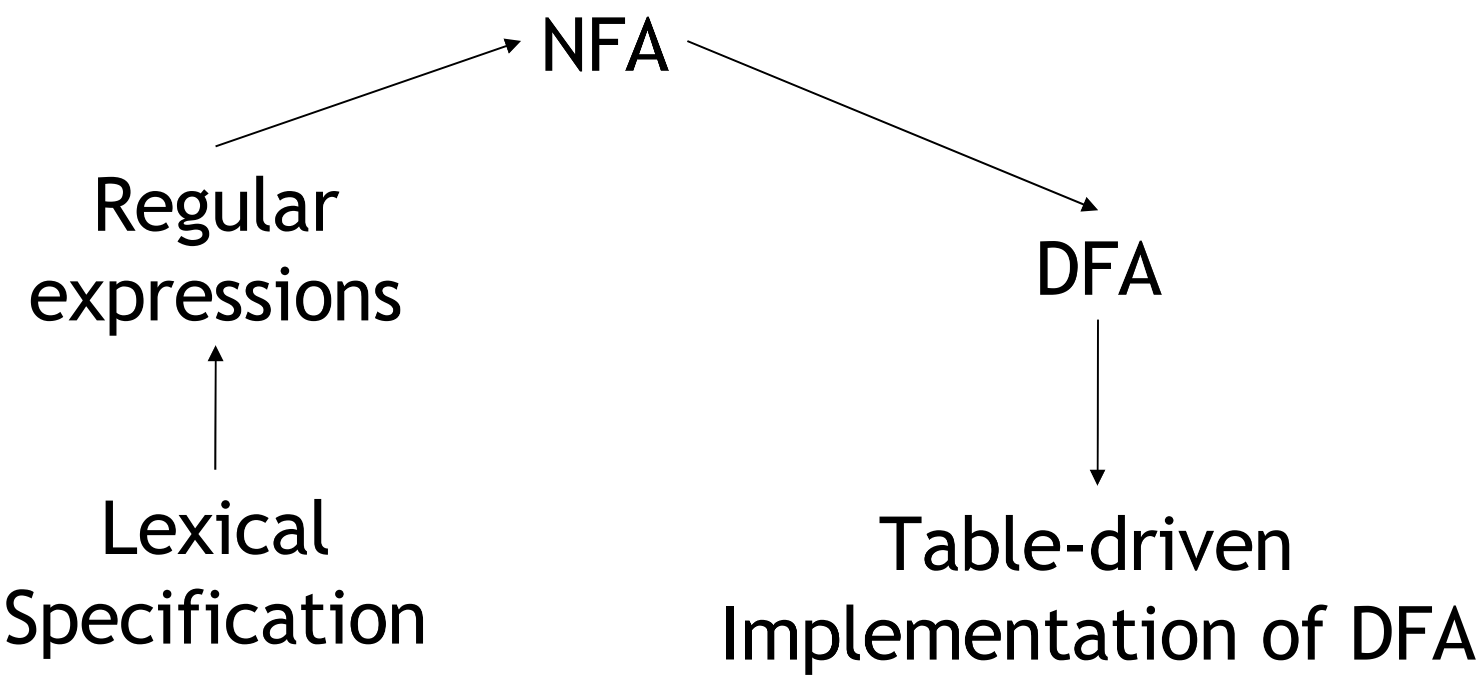
DFA



- DFA can be *exponentially* larger than NFA

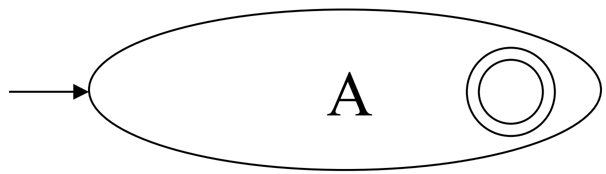
Regular Expressions to Finite Automata

- High-level sketch

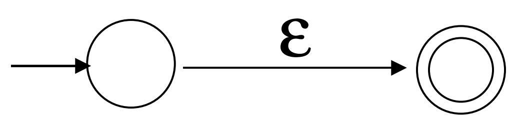


Regular Expressions to NFA (1)

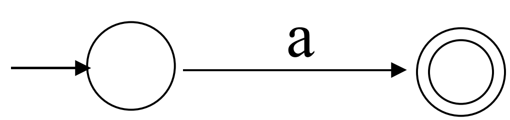
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



- For ε

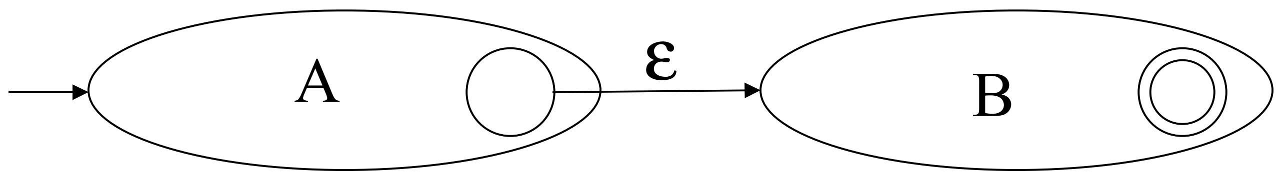


- For input a

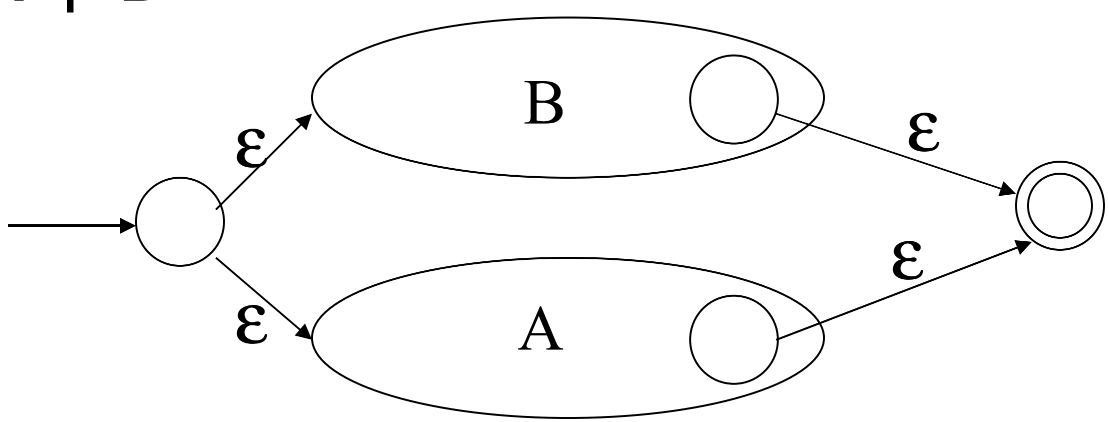


Regular Expressions to NFA (2)

- For AB

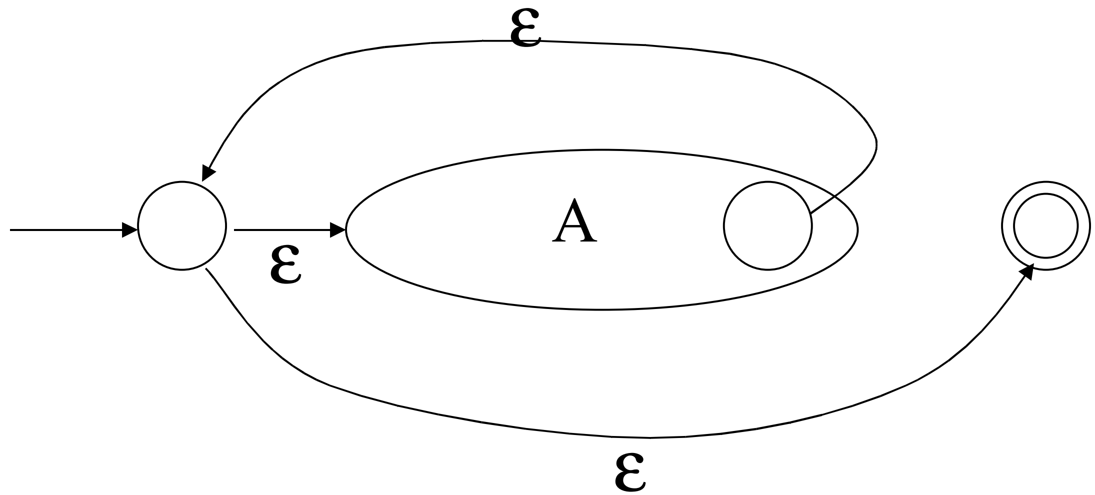


- For A | B



Regular Expressions to NFA (3)

- For A*

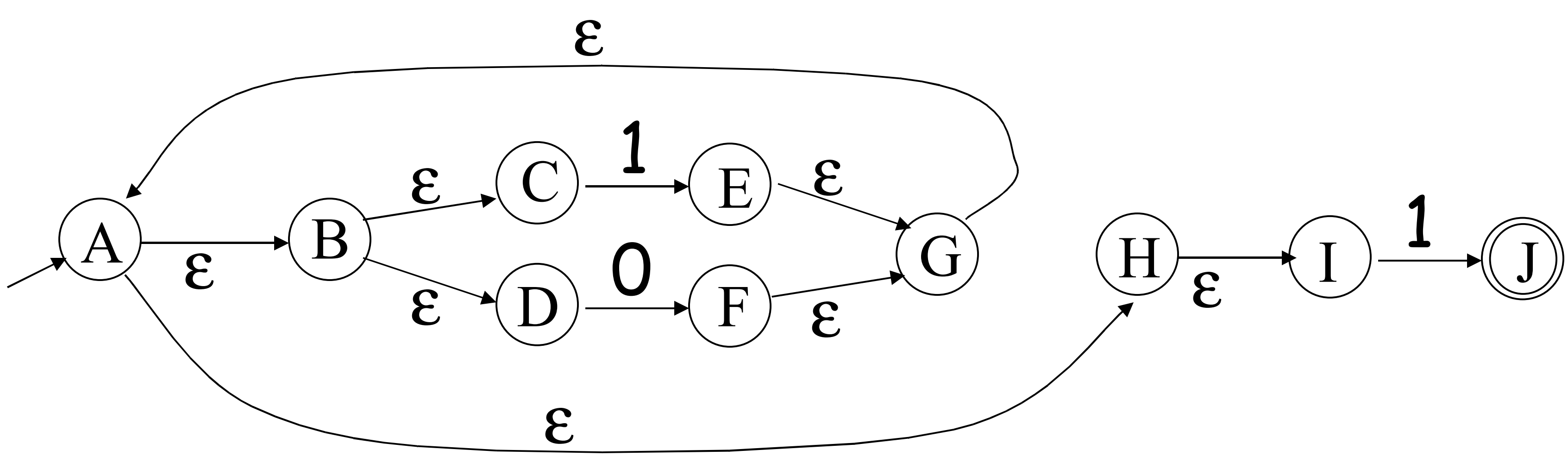


Example of RegExp -> NFA Conversion

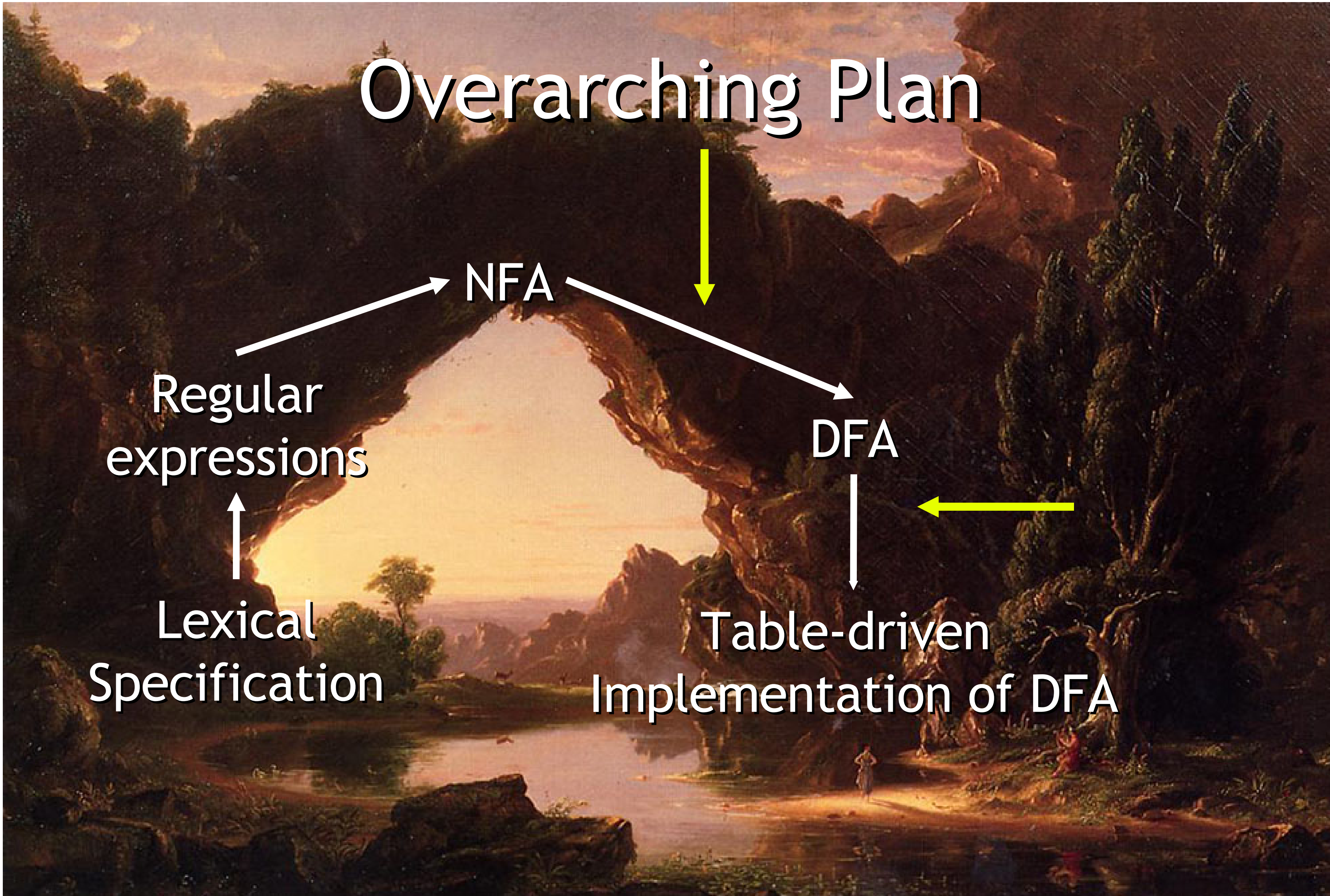
- Consider the regular expression

  (1 | 0)* 1

- The NFA is



Overarching Plan

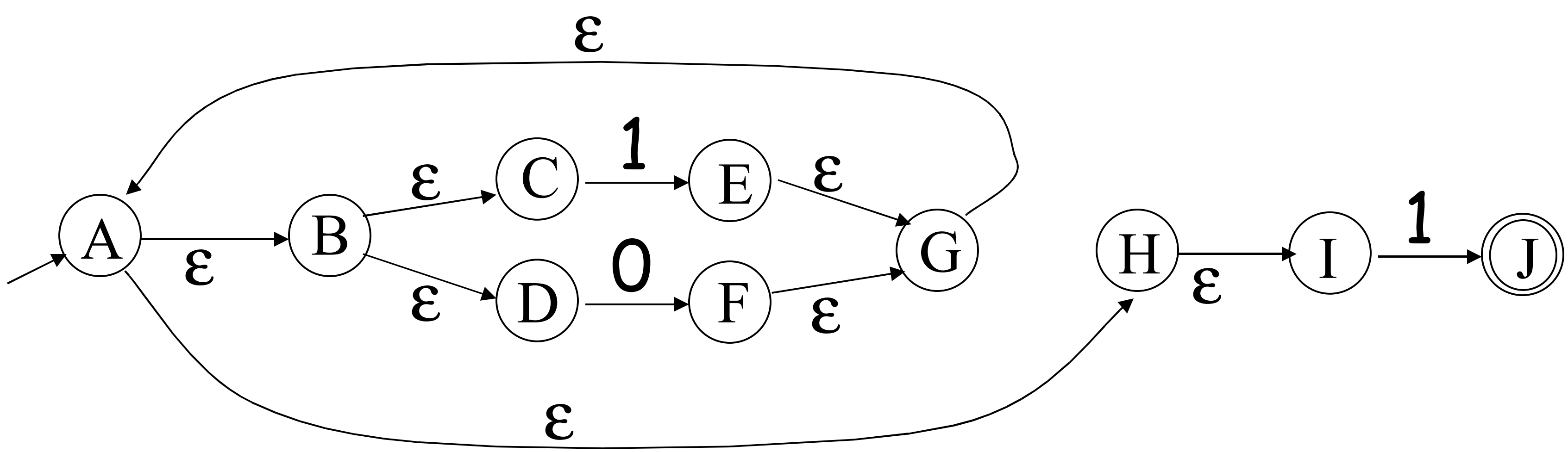


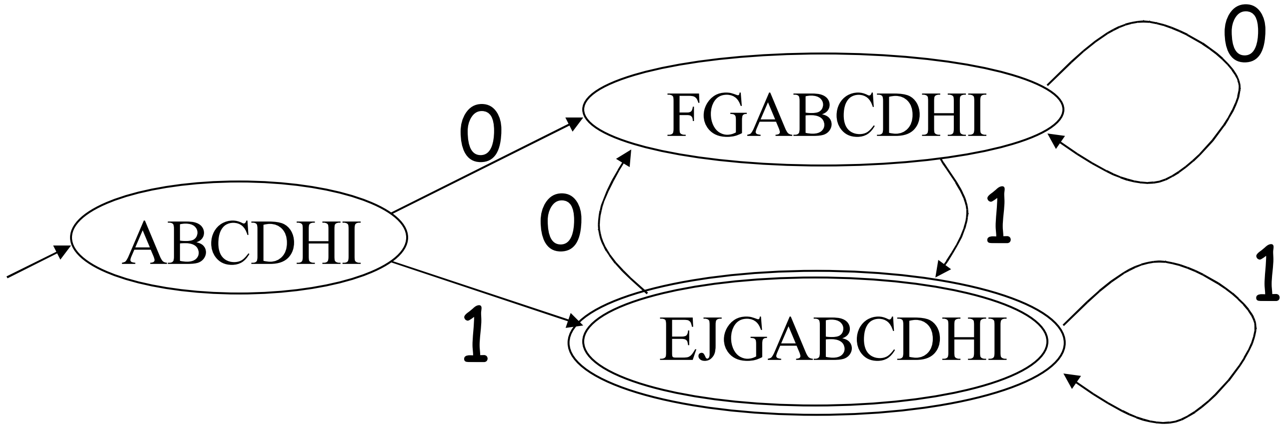
Thomas Cole – Evening in Arcady (1843)

NFA to DFA: The Trick

- Simulate the NFA
- Each state of 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





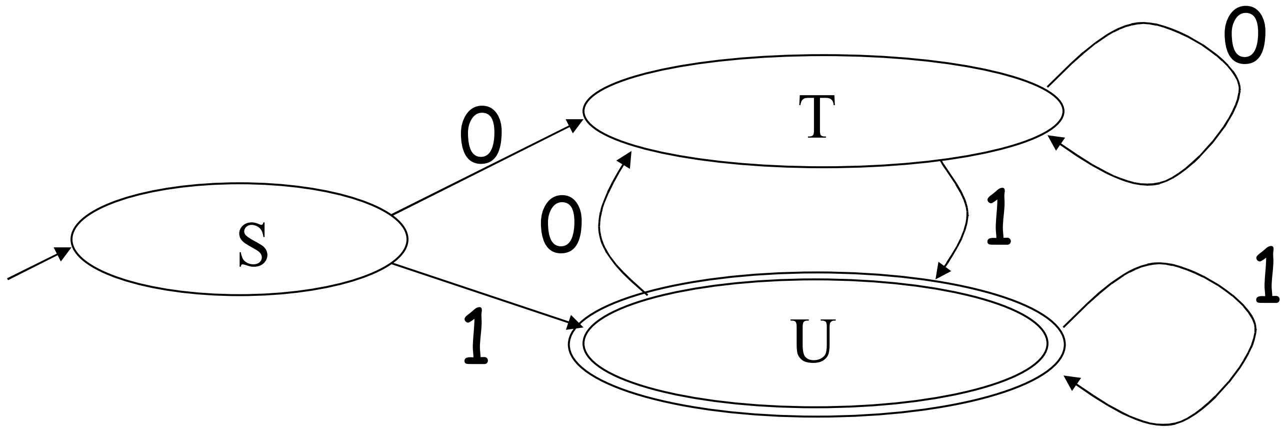
NFA $\rightarrow$ DFA: Remark

- 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

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \to^a S_k$ 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

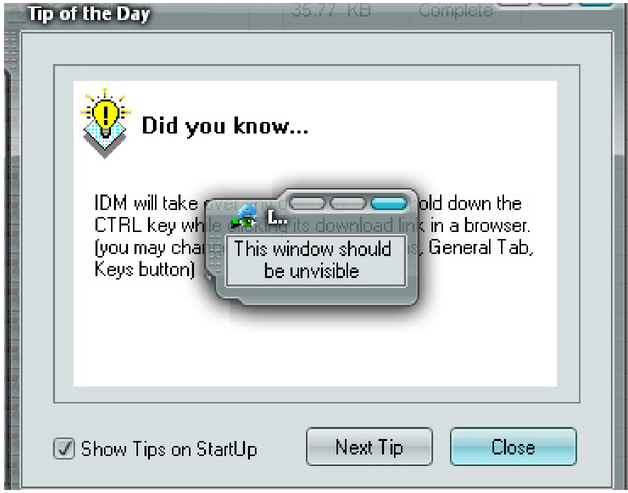


	0	1
S	T	U
T	T	U
U	T	U

Implementation (Cont.)

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

PA2: Lexical Analysis

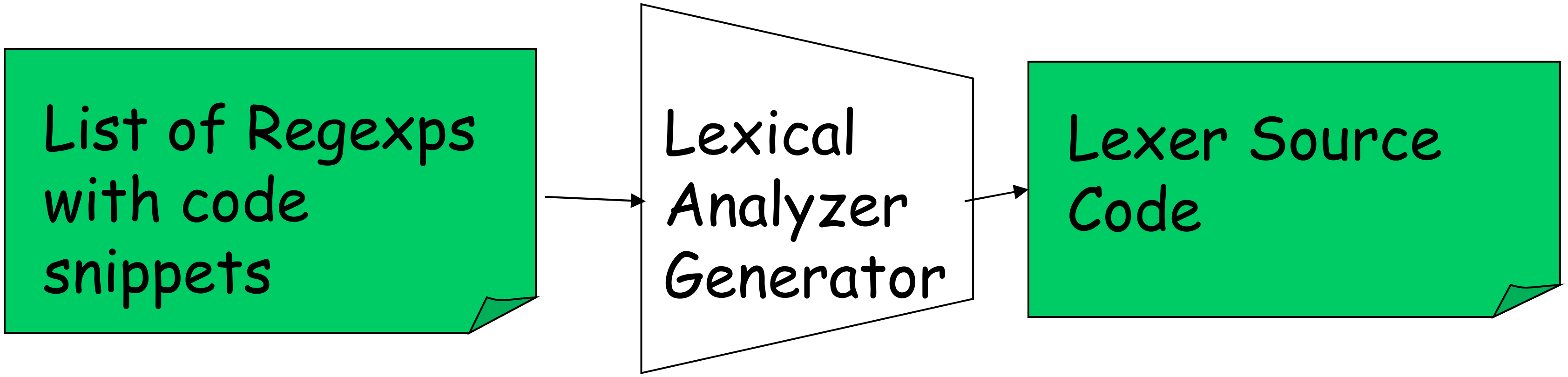


- Correctness is job #1.
  - And job #2 and #3!

- Tips on building large systems:
  - Keep it simple
  - Design systems that can be tested
  - Don’t optimize prematurely
  - It is easier to modify a working system than to get a system working

Lexical Analyzer Generator

- Tools like *lex* and *flex* and *ocamllex* will build lexers for you!
- You will use this for PA1



- I'll explain ocamllex; others are similar
  - See PA2 documentation

Ocamllex "lexer.mll" file

```
{
   (* raw preamble code
         type declarations, utility functions, etc. *)
}
let re_name_i = re_i
rule normal_tokens = parse
   re_1          { token_1 }
 | re_2          { token_2 }
and special_tokens = parse
 | re_n          { token_n }
```

Example “lexer.mll”

```
{
   type token = Tok_Integer of int        (* 123 *)
        | Tok_Divide                      (*  /  *)
}
let digit = [‘0’ - ‘9’]
rule initial = parse
   ‘/’           { Tok_Divide }
| digit digit* { let token_string = Lexing.lexeme lexbuf in
                 let token_val = int_of_string token_string in
                 Tok_Integer(token_val) }
| _            { Printf.printf “Error!\n”; exit 1 }
```

Adding Winged Comments

```
{
   type token = Tok_Integer of int        (* 123 *)
              | Tok_Divide                (*  /  *)
}
let digit = ['0' - '9']
rule initial = parse
   "//"            { eol_comment }
|  '/'             { Tok_Divide }
| digit digit*     { let token_string = Lexing.lexeme lexbuf in
                     let token_val = int_of_string token_string in
                     Tok_Integer(token_val) }
| _                { Printf.printf "Error!\n"; exit 1 }

and eol_comment = parse
  '\n'     { initial lexbuf }
| _        { eol_comment lexbuf }
```

Using Lexical Analyzer Generators

```
$ ocamllex lexer.mll
45 states, 1083 transitions, table size 4602 bytes

(* your main.ml file … *)
let file_input = open_in "file.cl" in
let lexbuf = Lexing.from_channel file_input in
let token = Lexer.initial lexbuf in
match token with
| Tok_Divide -> printf "Divide Token!\n"
| Tok_Integer(x) -> printf "Integer Token = %d\n" x
```

How Big Is PA2?

- The reference "lexer.mll" file is 88 lines
  - Perhaps another 20 lines to keep track of input line numbers
  - Perhaps another 20 lines to open the file and get a list of tokens
  - Then 65 lines to serialize the output
  - I'm sure it's possible to be smaller!
- Conclusion:
  - This isn't a code slog, it's about careful forethought and precision.

Homework

- Wednesday: PA1 due
- Thursday: Chapters 2.3 – 2.3.2
  - Optional Wikipedia article