Lexical Analysis

Implementation: Finite Automata

Outline

- Specifying lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)
- Implementation of regular expressions

  RegExp => NFA => DFA => Tables

Notation

- There is variation in regular expression notation

- Union: $A \mid B \equiv A + B$
- Option: $A + \varepsilon \equiv A?$
- Range: 'a' +' b' +...+' z' $\equiv$ [a-z]
- Excluded range:
  complement of [a-z] $\equiv$ [^a-z]

Regular Expressions in Lexical Specification

- Given a string s and a reg. exp. R, is

  $s \in L(R)$?

- But a yes/no answer is not enough!

- Instead: partition the input into tokens

- We adapt regular expressions to this goal

Regular Expressions => Lexical Spec.

1. Write a rexp for the lexemes of each token
   - Number = digit^{+} ~~+~~
   - Keyword = ‘if’+ ‘else’+ …
   - Identifier = letter (letter + digit)*
   - OpenPar = ‘(‘
   - …

Regular Expressions => Lexical Spec.

2. Construct R, matching all lexemes for all tokens

R = Keyword + Identifier + Number + …

= R_1+ R_2+ …

* So, if the lexeme doesn't belong to R, it's an error.

Lexeme $\notin L(R) \Rightarrow$ error

Regular Expressions => Lexical Spec.

3. Let input be $x_1 \ldots x_n$ all the characters of the program

   For $1 \leq i \leq n$ check

   $x_1 \ldots x_i \in L(R)$ substring from the input(program)

4. If success, then we know that

   $x_1 \ldots x_i \in L(R_j)$ for some j

5. Remove $x_1 \ldots x_i$ from input and go to (3)

it means that something could be intrepreted in more than one way

Ambiguities (1)

- There are ambiguities in the algorithm
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$ and also
  - $x_1 \ldots x_K \in L(R)$
- e.g. = and ==
- Rule: Pick longest possible string in $L(R)$
  - The "maximal munch"
  - We as humans do that.

Ambiguities (2)

- Which token is used? What if
  - $x_1 \ldots x_i \in L(R_j)$ and also
  - $x_1 \ldots x_i \in L(R_k)$

- e.g. 'if' could be an identifier or a keyword;
- which one to choose?
- Rule: use rule listed first (j if $j < k$)
  - Treats "if" as a keyword, not an identifier
- i.e. the one listed first is given higher priority

Error Handling

- What if
  - No rule matches a prefix of input ?
- Problem: Can't just get stuck …
- A compiler needs to give feedback to the user e.g. where the error is in the file (line number)
- Solution:

  reg-exp. that takes all strings (Σ^+), it will come to it only if the string doesn't match with any of the previous reg. langs..

  - Write a rule matching all "bad"strings
  - Put it last (lowest priority)

Summary

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

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 $state \rightarrow^{input} state$

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 and in accepting state => accept
- Otherwise => reject
  - If it terminates in state s that not a member of F
  - Or it gets stuck because there is not transition from state s1 on input a (i.e. never reaches the end of input).

Finite Automata State Graphs

- A state

- The start state

- An accepting state

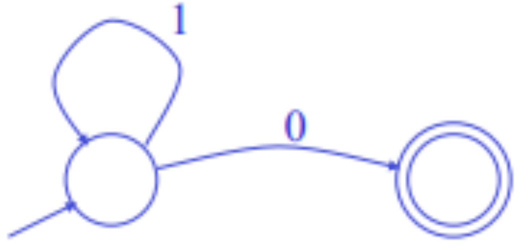
- A transition

A Simple Example

- A finite automaton that accepts only “1”

Another Simple Example

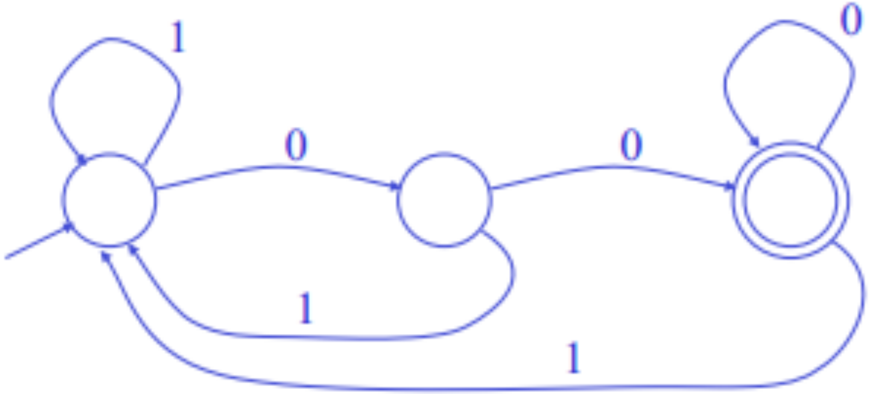
- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



And Another Example

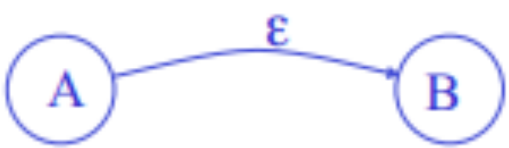
- Alphabet {0,1}
- What language does this recognize?

$(1+0)^*00^+$



Epsilon Moves

- Another kind of transition: ε-moves



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

Deterministic and Nondeterministic Automata

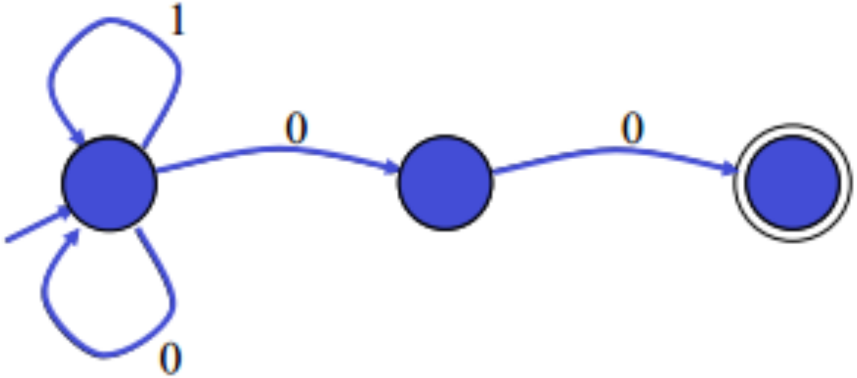
- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

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 0
- States: {A} {A,B} {A,B,C}
- Rule: NFA accepts if it can get to a final state

NFA vs. DFA

equivalent on power

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

- DFAs are faster to execute
  - There are no choices to consider

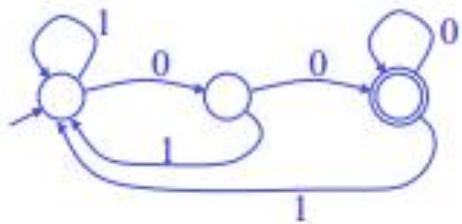
NFA vs. DFA

- For a given language NFA can be simpler than DFA

saves memory → NFA

Tradeoff

saves time → DFA



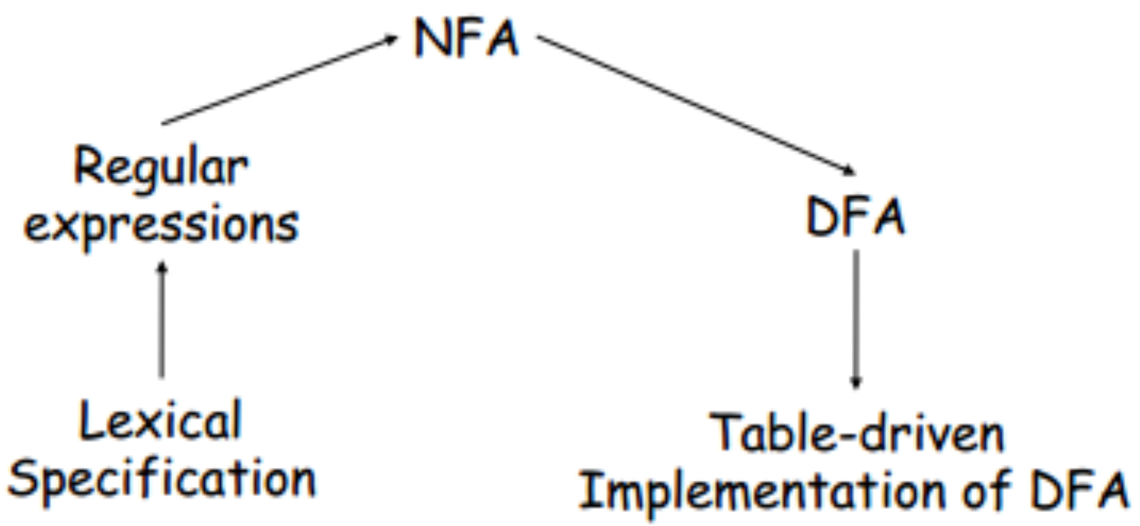
(worst case scenario)

- DFA can be exponentially larger than NFA

not always
sometimes NFA has
more states than equiv. DFA

Regular Expressions to Finite Automata

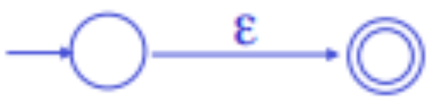
- High-level sketch



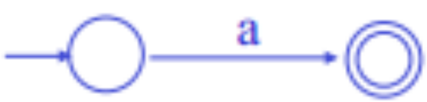
Regular Expressions to NFA (1)

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

- For ε

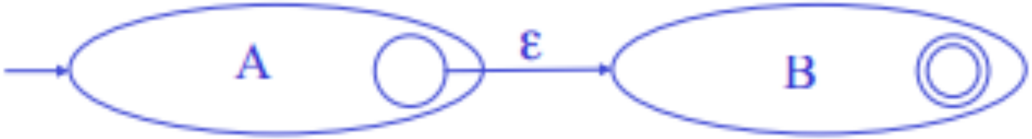


- For input a

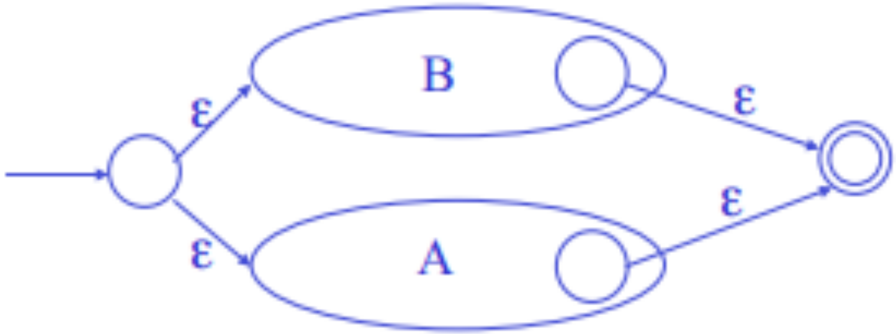


Regular Expressions to NFA (2)

- For AB

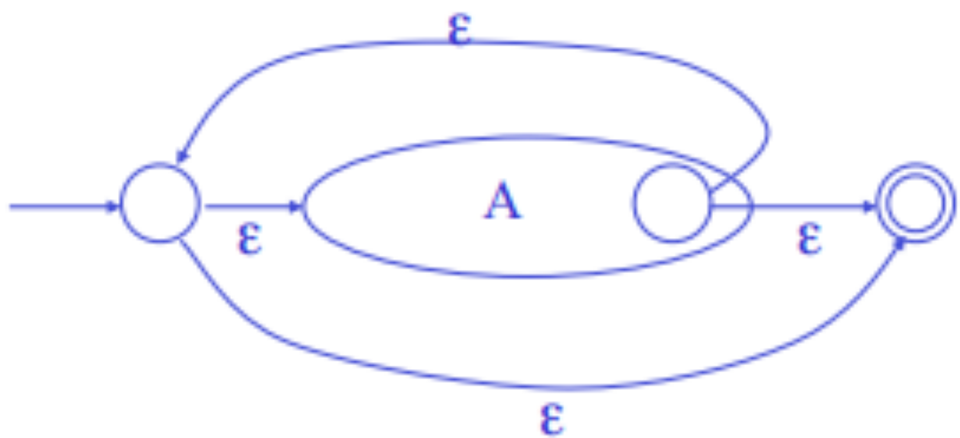


- For A + B



Regular Expressions to NFA (3)

- For *A**

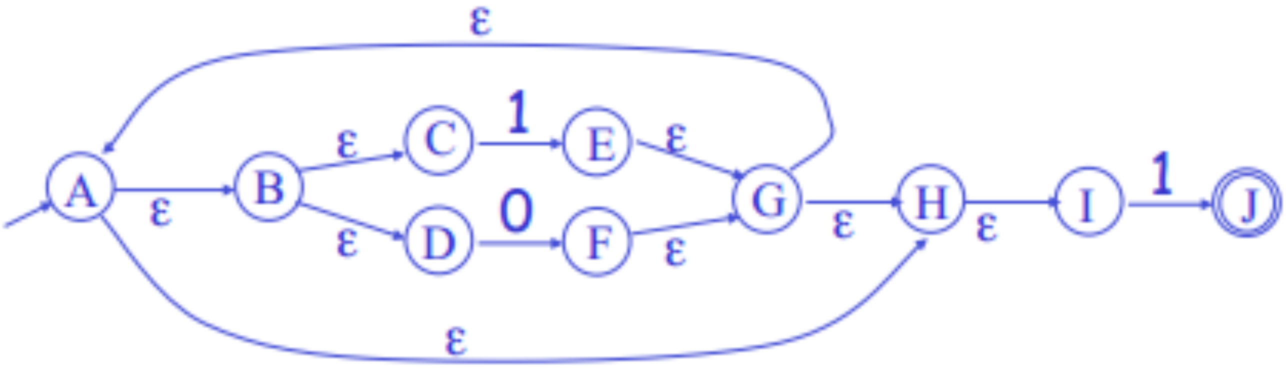


Example of RegExp -> NFA conversion

- Consider the regular expression

  (1+0)*1

- The NFA is



ε-closure of a state

- ε-closure of a state s is a set of states that consists of s and all other states that I can reach from s by making ε-moves only.
- Example
  - ε-closure(B) = {B, C, D}
  - ε-closure(G) = {G, H, I, A, B, C, D}

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 subsets are there?
  - 2^N-1 i.e., finitely many

to have the state in the subset or not (power set)

empty set

* In worst case scenario, I need to make each subset of NFA states as a state in DFA. (exponentially)

NFA	DFA
States : S	States : subset of S
Start state: $s \in S$	Start state: ϵ-closure(s)
Final states: F subset of S	Final state: $\{ X \mid X \cap F \neq \phi \}$ e.g. $\{EKRJS\} \cap \{J\} \neq \phi$
The transition function: subset of the states or maybe all of them. $a(x)=\{y \mid x \in X \wedge x \xrightarrow{a} y\}$ such that	The transition function: $X \xrightarrow{a} Y$ if $Y = \epsilon$-closure$(a(X))$

NFA to DFA: The Trick

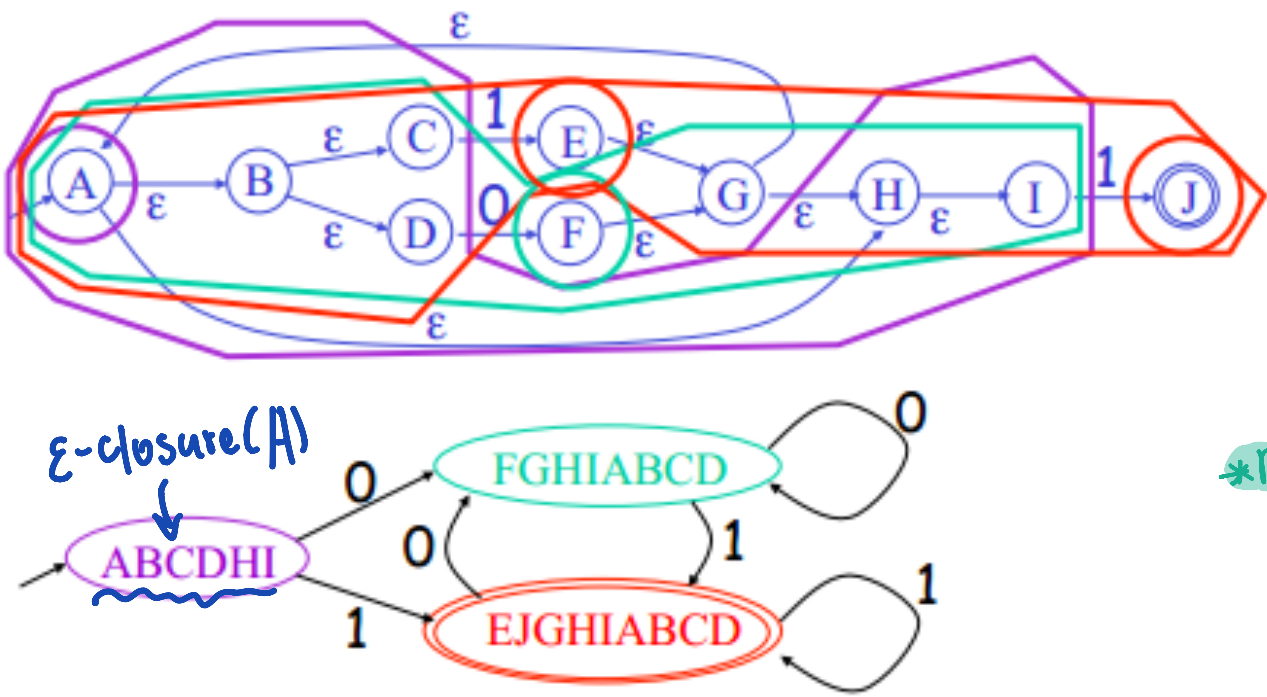
- Simulate the NFA
- Each state of DFA

  = 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 any state in S after seeing the input a, considering ε-moves as well

NFA -> DFA Example

$(1+0)^*1$



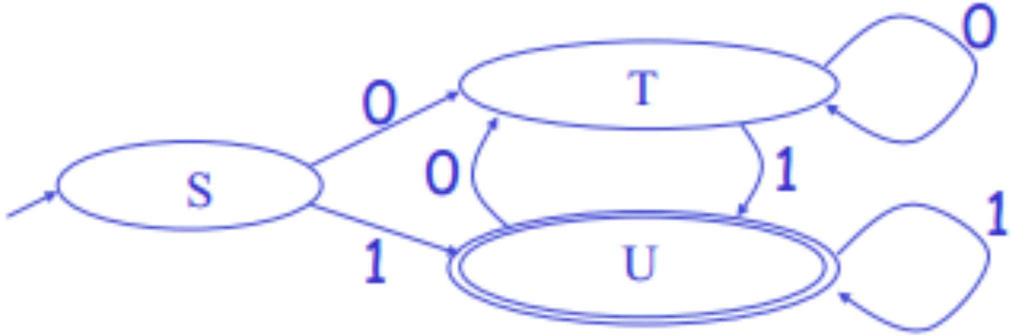
*recording

$\{EJGHIABCD\} \cap \{J\} \neq \phi = \{\}$

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbol"
  - 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



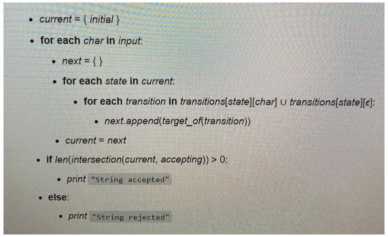
	0	1
S	T	U
T	T	U
U	T	U

algorithm

```
i=0;
State=0;
While(input[i]){
    State=A[state, input[i++]];
    }
```

→ if(state ∈ F) then
  Accept
else
  Reject

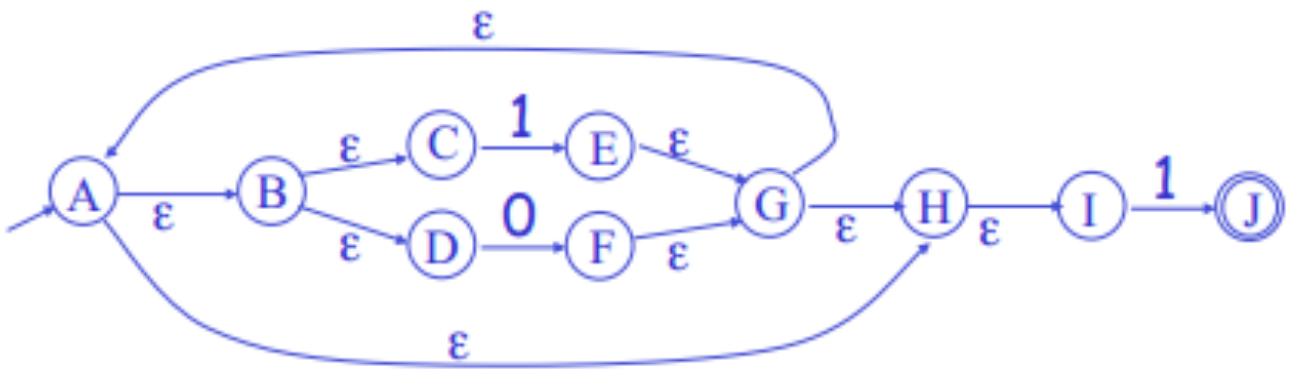
Taken from here: https://stackoverflow.com/questions/10626414/design-a-nondeterministic-finite-automata-in-c-incorrect-output



- *current* = { *initial* }
- **for each** *char* **in** *input*:
  - *next* = { }
  - **for each** *state* **in** *current*:
    - **for each** *transition* **in** *transitions*[*state*][*char*] ∪ *transitions*[*state*][*ϵ*]:
      - *next.append*(*target_of*(*transition*))
  - *current* = *next*
- **if** *len*(*intersection*(*current*, *accepting*)) > 0:
  - *print* `"String accepted"`
- **else**:
  - *print* `"String rejected"`

NFA Algorithm

Implementation of the NFA itself



	0	1	ξ
A			**{B}**
B			{ C,D }
C		{ E }	
D	{ F }		
:			

* We can track the exploration of paths by using graph search algorithm (Depth-First
Breadth-First).

Trade off between speed and space

- DFAs
  - time → – Faster: we are in one state at any given time.
  - memory → – less compact: there could be a large number of states 2^N-1.
- NFAs
  - slower (the loop has to deal with set of states rather than one state),
  - concise