Implementing Lexical Analyzers

Finite Automata

For lexical analysis:
- Specification — Regular expression
- Implementation — Finite automata

A **finite automata** consists of 5 components: (Σ, S, n, F, δ)
1. An input alphabet, Σ
2. A set of states, S
3. A start state, n $\in$ S
4. A set of accepting states F $\subseteq$ S
5. A set of transitions, $\delta : s_a \xrightarrow{input} s_b$

Finite Automata

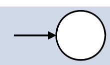
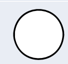
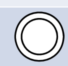
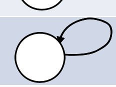
Transition $\delta : s_a \xrightarrow{input} s_b$

This is read as "In state S_a, go to state S_b, when input is encountered"

At the end of the input (or when no transition is possible), if in current state X
- If X $\in$ accepting set F, then **accept**
- otherwise, **reject**

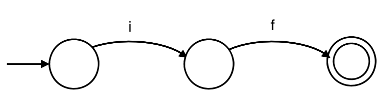
We sometimes prefer to use graphical representations of finite automata, known as a **state graph**.

State Graph Symbols

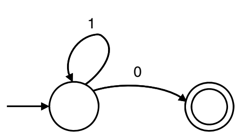
Start State	
State	
Accepting State	
Transition	
Self-loop	

Examples

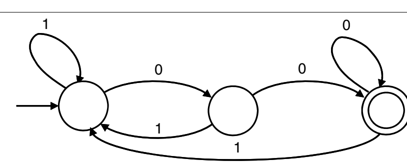
Alphabet = ASCII
Accepts: "if"



Alphabet = {0,1}
Accepts: 1*0



Examples



What language does this recognize? (Alphabet = {0,1})

Two or more 0s in a row at the end of the input

Regex: 00* *or* 00+ *or* 0{2,}

Table Implementation

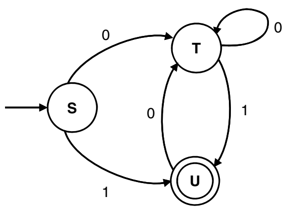
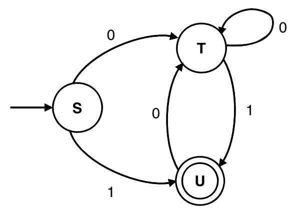


Table Implementation



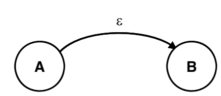
Input

State	0	1
S	T	U
T	T	U
U	T	X

Table-driven Code

```
FSA() {
        state = ‘S’;
        while (!done) {
                ch = fetch_input();
                state = Table[state][ch];
                if (state == ‘X’) {
                        System.err.println(“error”);
                }
        }
        if (state ∊ F){
                 System.out.println(“accept”);
        }
        else {
                 System.out.println(“reject”);
        }
}
```

Epsilon Transitions



Another kind of transition: ε- transition

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

DFA & NFAs

Deterministic Finite Automata (DFA):

- One transition per input per state
- No ε-moves

Non-deterministic Finite Automata (NFA):

- Can have multiple transitions for one input in a given state
- Can have ε-moves

Finite automata have finite memory

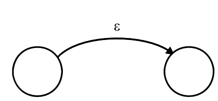
- Need only to encode the current state

Converting REs to NFAs

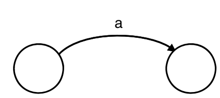
Thompson’s Algorithm

REs can be converted to NFAs. Atomic REs are straightforward.

Epsilon transitions:

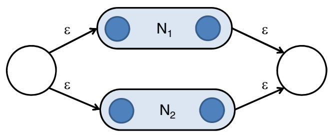


Single characters:

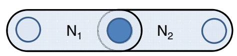


Converting REs to NFAs

Alternation:
$N_1 \mid N_2$

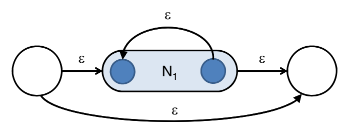


Concatenation:
$N_1 N_2$



Converting REs to NFAs

Kleene Closure:
N_1^*



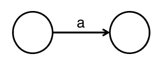
Example

Convert (a | b) * ab to an NFA

Example

Convert (a | b) * ab to an NFA

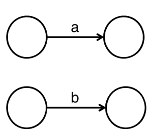
Step 1: a



Example

Convert (a | b) * ab to an NFA

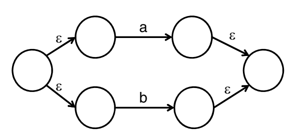
Step 2: b



Example

Convert (a | b) * ab to an NFA

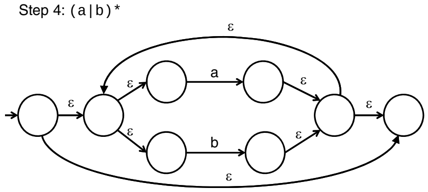
Step 3: (a | b)



Example

Convert `(a|b)*ab` to an NFA

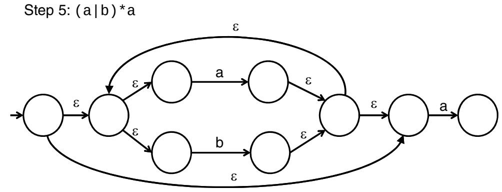
Step 4: `(a|b)*`



Example

Convert `(a|b)*ab` to an NFA

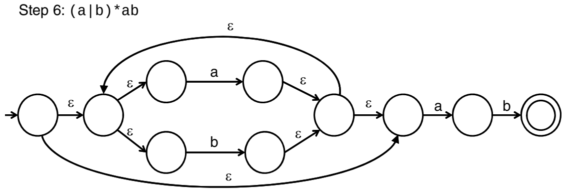
Step 5: `(a|b)*a`



Example

Convert `(a|b)*ab` to an NFA

Step 6: `(a|b)*ab`



Executing Finite Automata

A DFA can take only one path through the state graph

- Completely determined by input

A NFA can take multiple paths "simultaneously"

- NFAs make ε-transitions
- There may be multiple transitions out of a state for a single input
- **Rule**: the NFA accepts it if can get into a final state by any path

Which is more powerful, an NFA or a DFA?

Power of NFAs and DFAs

Theorem: NFAs and DFAs recognize the same set of languages

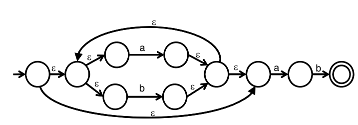
Both recognize regular languages.

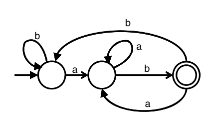
DFAs are faster to execute because there are no choices to consider.

For a given language, the NFA can be simpler than the DFA – a DFA can be exponentially larger.

Example

NFA and DFA that accept `(a|b)*ab`





NFA to DFA Conversion

Basic idea: Given a NFA, simulate its execution using a DFA

- At step *n*, the NFA may be in any of multiple possible states

The new DFA is constructed as follows:

- The states of the DFA correspond to a non-empty subset of states of the NFA
- The DFA's start state is the set of NFA states reachable through ε-transitions from NFA start state
- A transition $S_a \xrightarrow{c} S_b$ is added **iff** S_b is the set of NFA states reachable from any state in S_a after seeing the input *c*, also considering ε-transitions

Epsilon-Closure

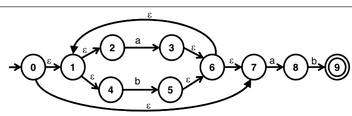
Let **edge**(*s*,*c*) be the set of all NFA states reachable by following a single edge with label *c* from state *s*.

For a set of states S, **ε-closure**(S) is the set of states that can be reached from a state in S via ε-transitions.

$$\varepsilon\text{−}Closure(S) = S \cup \left(\bigcup_{s \in T} edge(s,\varepsilon)\right)$$

function ε-closure(S)
  T ← S
  repeat
    T' ← T
    $T = T' \cup (\cup_{s \in T'} \mathrm{edge}(s, \varepsilon))$
  until T=T'
  return T

Start State



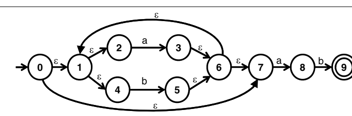
The NFA's start state is S_0, so the DFA's start state = ε-closure(S_0)

By iteration:

$T_1 = S_0 = \{S_0\}$
$T_2 = T_1 \cup$ ε-closure(T_1) $= \{S_0, S_1, S_7\}$
$T_3 = T_2 \cup$ ε-closure(T_2) $= \{S_0, S_1, S_2, S_4, S_7\}$
$T_4 = T_3 \cup$ ε-closure(T_3) $= \{S_0, S_1, S_2, S_4, S_7\}$

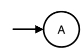
$T_4 = T_3$ so we are done.

NFA to DFA Conversion Example



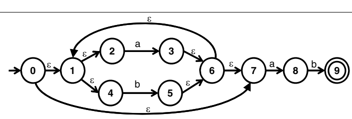
Start state = ε-closure(S_0) = {0, 1, 2, 4, 7} = A

We'll call this collection of states A, and will be a new node in our DFA that is our DFA start state.



Set	Name
{0, 1, 2, 4, 7}	A

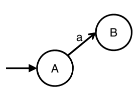
Construct DFA



We now compute where we can go from A on each input in our alphabet.

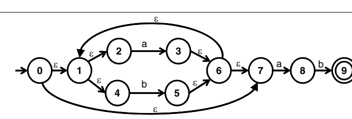
On an 'a', considering each state in A, where might we end up? An a would take us from 2 to 3 and from 7 to 8. But we must consider our ε-transitions as well.

B = ε-closure(3) ∪ ε-closure(8) = {1, 2, 3, 4, 6, 7 } ∪ {8}



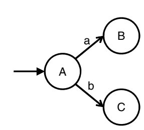
Set	Name
{0, 1, 2, 4, 7}	A
{1, 2, 3, 4, 6, 7, 8 }	B

Construct DFA



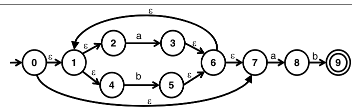
On an 'b', considering each state in A, we could go to 5, but we must do the ε-closure.

C = ε-closure(5) = {1, 2, 4, 5, 6, 7}



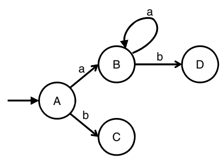
Set	Name
{0, 1, 2, 4, 7}	A
{1, 2, 3, 4, 6, 7, 8 }	B
{1, 2, 4, 5, 6, 7}	C

Construct DFA



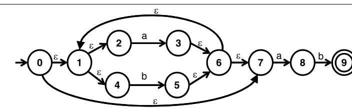
Repeat process for B:

In B, see an 'a' = {1, 2, 3, 4, 6, 7, 8} = B (Self loop)
In B, see a 'b' = {1, 2, 4, 5, 6, 7, 9} = D



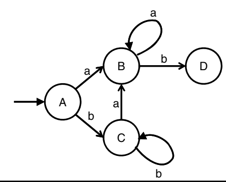
Set	Name
{0, 1, 2, 4, 7}	A
{1, 2, 3, 4, 6, 7, 8 }	B
{1, 2, 4, 5, 6, 7}	C
{1, 2, 4, 5, 6, 7, 9}	D

Construct DFA



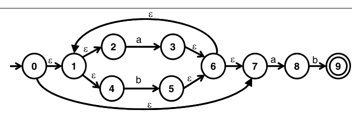
Repeat process for C:

In C, see an 'a' = {1, 2, 3, 4, 6, 7, 8} = B
In C, see a 'b' = {1, 2, 4, 5, 6, 7} =C (Self loop)



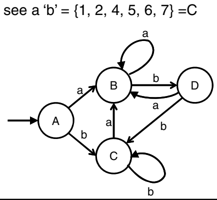
Set	Name
{0, 1, 2, 4, 7}	A
{1, 2, 3, 4, 6, 7, 8 }	B
{1, 2, 4, 5, 6, 7}	C
{1, 2, 4, 5, 6, 7, 9}	D

Construct DFA



Repeat process for D:

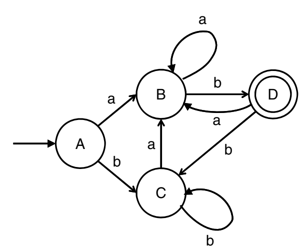
In D, see an 'a' = {1, 2, 3, 4, 6, 7, 8} = B
In D, see a 'b' = {1, 2, 4, 5, 6, 7} =C



Set	Name
{0, 1, 2, 4, 7}	A
{1, 2, 3, 4, 6, 7, 8 }	B
{1, 2, 4, 5, 6, 7}	C
{1, 2, 4, 5, 6, 7, 9}	D

DFA Final States

A state in the DFA is final if one of the states in the set of NFA states is final.



Set	Name
{0, 1, 2, 4, 7}	A
{1, 2, 3, 4, 6, 7, 8 }	B
{1, 2, 4, 5, 6, 7}	C
{1, 2, 4, 5, 6, 7, 9}	D

NFA to DFA Remarks

This algorithm does not produce a minimal DFA.

It does however, exclude states that are not reachable from the start state.

This is important because an n-state NFA could have 2^n states as a DFA. (Why? Set of all subsets.)

The minimization algorithm is left to the graduate course.

Why DFAs?

Why'd we do all that work?

A DFA can be implemented by a 2D table T:

- One dimension is states, the other dimension is input characters
- For $S_a \xrightarrow{c} S_b$ we have $T[S_a,c] = S_b$

DFA execution:

- If the current state is S_a and input is c, then read $T[S_a,c]$
- Update the current state to S_b, assuming $S_b = T[S_a,c]$
- This is very efficient

Automating Automatons

If we have algorithmic ways to convert REs to NFAs and to convert NFAs to faster DFAs, we could have a program where we write our lexical rules using REs and automatically have a table-driven lexer produced.

NFA to DFA conversion is the heart of automated tools such as **lex/flex/JLex/Jflex**

- DFA could be very large
- In practice, lex-like tools trade off speed for space in the choice of NFA and DFA representations

Implementation

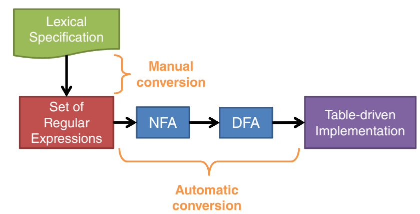
RE → NFA → DFA → Table-driven Implementation

- Specify lexical structure using regular expressions

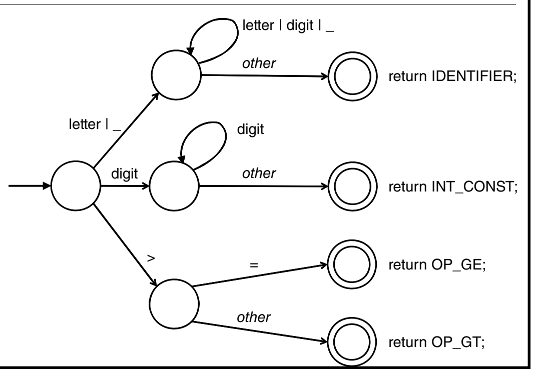
Finite automata

- Deterministic Finite Automata (DFAs)
- Non-deterministic Finite Automata (NFAs)

Table implementation



Scanner Automaton



Ambiguity Resolution

Imagine a rule for C identifiers:
`[a-zA-Z_][a-zA-Z0-9_]*`

And the rule for a keyword such as if:
`“if”`

How do we resolve the fact that `if` is a keyword and `if8` is an identifier?

Two rules:

1. **Longest match** – The match with the longest string will be chosen.
2. **Rule priority** – for two matches of the same length, the first regex will be chosen. I.e., Rule order matters.