Automata Theory and Formal Grammars: Lecture 3

Regular Expressions and Languages

Regular Expressions and Languages

Last Time

- Deterministic Finite Automata (DFAs) and their Languages
- Closure Properties of DFA Languages (the product construction)
- Nondeterministic Finite Automata (NFAs) and their Languages
- Relating DFAs and NFAs (the subset construction)

Today

- Regular Expressions and Regular Languages
- Properties of Regular Languages
- Relating NFAs and regular expressions: Kleene's Theorem

NFAs: Finishing Up

NFAε

Sipser uses a more general definition than I gave last week:

Definition A nondeterministic finite automaton with empty transitions (NFAε) is a quintuple $\langle Q, \Sigma, q_0, \delta, A\rangle$ where:

- Q is a finite set of states;
- Σ is the input alpabet;
- $q_0 \in Q$ is the start state;
- $A \subseteq Q$ is the set of accepting states; and
- $\delta : Q \times \Sigma \cup \{\varepsilon\} \rightarrow 2^Q$ is the transition function.

Relating NFA and NFAε

Theorem The set of NFA languages is identical to the set of NFAε languages.

Proof?

One direction is trivial: An NFA (without empty transitions) is an NFAε where for all q:

$$\delta(q, \varepsilon) = \emptyset$$

The Subset Construction for NFAε

Let $N = \langle Q, \Sigma, q_0, \delta, A \rangle$ be a NFAε.
We want to construct a DFA $D(N)$ accepting the same language.
States in $D(N)$ will be sets of states from N.
Let P range over states of $D(N)$.
$P \in 2^Q$, that is, $P \subseteq Q$.

$$D(N) = \langle 2^Q \, , \, \Sigma \, , \, \delta(q_0, \varepsilon) \, , \, \delta_{DN} \, , \, A_{DN} \rangle$$

where

$$\delta_{DN}(P, a) = \bigcup_{q \in P} \delta^*(q, a)$$

$$A_{DN} = \{ \, P \mid P \in 2^Q \text{ and } P \cap A \neq \emptyset \, \}$$

Example

Consider the NFA M given by $K = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, 2\}$, $s = q_0$, $F = \{q_2\}$ with transition relation Δ given below:

q	σ	Δ(q,σ)
q_0	0	q_0
q_0	ε	q_1
q_1	1	q_1
q_1	ε	q_2
q_2	2	q_2

$\mathcal{L}(M) = \{0\}^*\{1\}^*\{2\}^*$.

Example continued

The resulting DFA M' has $K' = \{\{q_0, q_1, q_2\}, \{q_1, q_2\}, \{q_2\}, \emptyset\}$, $s' = \{q_0, q_1, q_2\}$, $F = \{\{q_0, q_1, q_2\}, \{q_1, q_2\}, \{q_2\}\}$ and δ':

q	σ	δ' (q,σ)
$\{ q_0, q_1, q_2 \}$	0	$\{ q_0, q_1, q_2 \}$
$\{ q_0, q_1, q_2 \}$	1	$\{ q_1, q_2 \}$
$\{ q_0, q_1, q_2 \}$	2	$\{ q_2 \}$
$\{ q_1, q_2 \}$	0	$\emptyset$
$\{ q_1, q_2 \}$	1	$\{ q_1, q_2 \}$
$\{ q_1, q_2 \}$	2	$\{ q_2 \}$
$\{ q_2 \}$	0,1	$\emptyset$
$\{ q_2 \}$	2	$\{ q_2 \}$
$\emptyset$	0,1,2	$\emptyset$

Another example

Let $\Sigma = \{a_1, ..., a_n\}$ where $n \geq 2$.

Let $L = \{\, w \mid \exists i.\ a_i$ does not appear in $w \,\}$.

For example, If $\Sigma = \{a_1, a_2, a_3\}$ then $a_1a_1a_2 \in \Sigma$ but $a_1a_2a_3 \notin \Sigma$.

Intuitively, the NFA would work in the following manner:

- Guess the symbol a_i that is missing from the input.
- If no symbol is missing, move to a dead state.
- If a symbol a_i is missing, go to state q_i.
- If in state q_i you ever encounter a_i, move to a dead state.
- Otherwise eat the remaining symbols and accept.

Another Example (continued)

For the construction of the NFA we need one starting state q_0 and one state for each symbol in the alphabet, $q_1, \ldots, q_n$.

There are ε-transitions from q_0 into each of $q_1, \ldots, q_n$, and self-loops on each of $q_1, \ldots, q_n$ labeled with the states that are legal.

What happens when we use the construction to produce a DFA accepting this language?

The equivalent DFA M' has initial state $\underline{s}' = \{q_0, q_1, q_2, q_3, ..., q_n\}$.

Regular Languages

Regular Languages

This course: a study of the computing power needed to “process” different kinds of languages.

The first class of languages we will study: regular languages.

Regular languages are defined using regular expressions.

Regular Expressions

... a notation for defining languages.

Definition Let Σ be an alphabet. Then the set $\mathcal{R}(\Sigma)$ of regular expressions over Σ is defined recursively as follows.

$$\begin{aligned}
\underline{\emptyset} &\in \mathcal{R}(\Sigma) \\
\underline{\varepsilon} &\in \mathcal{R}(\Sigma) \\
\underline{a} &\in \mathcal{R}(\Sigma) \quad \text{if } a \in \Sigma \\
\underline{r + s} &\in \mathcal{R}(\Sigma) \quad \text{if } \underline{r} \in \mathcal{R}(\Sigma) \text{ and } \underline{s} \in \mathcal{R}(\Sigma) \\
\underline{r \circ s} &\in \mathcal{R}(\Sigma) \quad \text{if } \underline{r} \in \mathcal{R}(\Sigma) \text{ and } \underline{s} \in \mathcal{R}(\Sigma) \\
\underline{r*} &\in \mathcal{R}(\Sigma) \quad \text{if } \underline{r} \in \mathcal{R}(\Sigma)
\end{aligned}$$

Comments about Regular Expressions

The previous definition just gives the syntax of regular expressions: $\underline{\circ}, \underline{\cup}, \underline{*}$ are symbols that we will shortly give an interpretation to.

Examples Let $\Sigma = \{a, b\}$. The following are regular expressions in $\mathcal{R}(\Sigma)$.

- $\underline{a}$
- $\underline{(a + (b \circ b))*}$
- $\underline{(((b*) \circ ((a \circ a) + b)) \circ \emptyset)}$

Notation

Usually, $\underline{\circ}$ will be omitted.

Also, to reduce parentheses, we will adopt the following precedence: $\underline{*} > \underline{\circ} > \underline{\cup}$.

So $\underline{(((b*) \circ ((a \circ a) + b)) \circ \emptyset)}$ can be written as $\underline{b*(aa + b)\emptyset}$.

Derived Operations

We will sometimes use the following derived operations on regular expressions.

$$\underline{r^+} = \underline{r \circ (r*)}$$

$$\underline{r^i} = \begin{cases} \underline{\varepsilon} & \text{if } i = 0 \\ \underline{r \circ (r^{i-1})} & \text{otherwise} \end{cases}$$

E.g. $\underline{(b + a)^2} = \underline{(b + a) \circ (b + a) \circ \varepsilon}$

How Do Regular Expressions “Define” Languages?

To make connection with languages precise, we need to define a semantics for regular expressions saying what they “mean”.

- Semantics will be given in form of function $\mathcal{L} : \mathcal{R}(\Sigma) \to 2^{\Sigma^*}$.
- For any regular expression $\underline{r}$, $\mathcal{L}(\underline{r}) \subseteq \Sigma^*$ will be the language defined by $\underline{r}$.

The Semantics of Regular Expressions

Definition Fix alphabet Σ. Then $\mathcal{L} : \mathcal{R}(\Sigma) \to 2^{\Sigma^*}$ is defined as follows.

$$\mathcal{L}(r) = \begin{cases} \emptyset & \text{if } r = \emptyset \\ \{\varepsilon\} & \text{if } r = \varepsilon \\ \{a\} & \text{if } r = a \text{ and } a \in \Sigma \\ \mathcal{L}(s_1) \cup \mathcal{L}(s_2) & \text{if } r = s_1 + s_2 \\ \mathcal{L}(s_1) \circ \mathcal{L}(s_2) & \text{if } r = s_1 \circ s_2 \\ (\mathcal{L}(s))^* & \text{if } r = s* \end{cases}$$

Definition $L \subseteq \Sigma^*$ is a regular language if there is a regular expression r such that $L = \mathcal{L}(r)$.

(Note: This is a denotational semantics.)

Questions about Regular Languages

1. What language does $(a+b)*$ define?
   All strings built from a and b.

$$\begin{aligned} \mathcal{L}((a+b)*) &= (\mathcal{L}(a+b))^* \\ &= (\mathcal{L}(a) \cup \mathcal{L}(b))^* \\ &= (\{a\} \cup \{b\})^* \;=\; \{a,b\}^* \end{aligned}$$

2. What is $\mathcal{L}(((a+b)(a+b))*)$?
   All even-length strings from $\{a,b\}^*$.

$$\begin{aligned} \mathcal{L}(((a+b)(a+b))*) &= (\mathcal{L}((a+b)(a+b)))^* \\ &= (\mathcal{L}(a+b) \circ \mathcal{L}(a+b))^* \\ &= (\{a,b\} \circ \{a,b\})^* \\ &= \{aa, ab, ba, bb\}^* \end{aligned}$$

Questions (cont.)

Let $\Sigma = \{a,b\}$.

1. What is a regular expression for all words in Σ^* ending in a?
   $(a+b)*a$
2. What is a regular expression for all odd-length words in Σ^*?
   $((a+b)(a+b))*(a+b)$
3. How do you prove that L_1 comprising words with exactly two b's is regular?
   Give a regular expression r_1 such that $\mathcal{L}(r_1) = L_1$. One choice for r_1 is $a*ba*ba*$.
4. How do you prove that L_2 consisting of words not containing ab is regular?
   Give a regular expression r_2 such that $\mathcal{L}(r_2) = L_2$. One choice for r_2 is $b*a*$.

Simplifying Regular Expressions

Definition Let r_1, r_2 be regular expressions. Then $r_1 =_{\mathcal{L}} r_2$ exactly when $\mathcal{L}(r_1) = \mathcal{L}(r_2)$.

Some Laws for $=_{\mathcal{L}}$

$$\begin{aligned} r + \emptyset &=_{\mathcal{L}} r \\ r \circ \emptyset &=_{\mathcal{L}} \emptyset \circ r \;=_{\mathcal{L}}\; \emptyset \\ r \circ \varepsilon &=_{\mathcal{L}} \varepsilon \circ r \;=_{\mathcal{L}}\; r \\ r_1 \circ (r_2 \circ r_3) &=_{\mathcal{L}} (r_1 \circ r_2) \circ r_3 \\ r_1 \circ (r_2 + r_3) &=_{\mathcal{L}} (r_1 \circ r_2) + (r_1 \circ r_3) \\ (r+s)* &=_{\mathcal{L}} r* \text{ if } \mathcal{L}(s) \subseteq \mathcal{L}(r*) \\ (r+\varepsilon)* &=_{\mathcal{L}} r* \end{aligned}$$

Finite Languages and Regularity

$\boxed{\text{Definition}}$ A language L is finite if it contains a finite number of words.

$\boxed{\text{Example}}$ $L_1 = \{aa, b, aba\}$ is finite; $L_2 = \{\, w \in \{a,b\}^* \mid |w| \text{ is even} \,\}$ is not.

It turns out that every finite language is regular!

$\boxed{\text{E.g.}}$ Regular expression for L_1 is $\underline{aa + b + aba}$.

A proof of this fact would use induction (on what?) and might rely on a lemma ("subtheorem") about singleton languages.

Singleton Languages are Regular

$\boxed{\text{Lemma}}$ For any $w \in \Sigma^*$, the language $\{w\}$ is regular.

Proof: Define the function $f_{word} : \Sigma^* \to \mathcal{R}(\Sigma)$ as follows

$$f_{word}(w) = \begin{cases} \underline{\varepsilon} & \text{when } w = \varepsilon \\ \underline{ar'} & \text{when } w = aw' \text{ and } f_{word}(w') = \underline{r'} \end{cases}$$

By induction on w, show that for all w, $\mathcal{L}(f_{word}(w)) = \{w\}$.

For a more detailed version, see the following slides.

Finite Languages are Regular

$\boxed{\text{Lemma}}$ Any finite language is regular.

Proof: Define the relation $f_{lang} \subset 2^{\Sigma^*} \to \mathcal{R}(\Sigma)$ as follows

$$f_{lang}(L) = \begin{cases} \underline{\emptyset} & \text{when } L = \emptyset \\ \underline{r + r'} & \text{when } L = \{w\} \cup L' \text{ and } f_{word}(w) = \underline{r} \\ & \qquad \text{and } f_{lang}(L') = \underline{r'} \end{cases}$$

By induction on $k \in \mathbb{N}$, show that if $|L| = k$ then $\mathcal{L}(f_{lang}(L)) = L$.

Note that f_{lang} is not actually a well defined function.
We need to pick words $\{w\}$ deterministically.
This can be done by defining an order on regular expressions.

Singleton Languages Detail (1)

$\boxed{\text{Lemma}}$ Let Σ be an alphabet, and let $w \in \Sigma^*$. Then the language $\{w\}$ is regular.

How do we prove this? First, write down the logical form.

$\boxed{\text{Logical Form}}$ $\forall w \in \Sigma^*.\ P(w)$, where $P(w)$ is "$\{w\}$ is regular."

We can prove this by induction on the definition of Σ^*; i.e. we could prove the statement $\forall k \in \mathbb{N}.\ \forall w \in (\Sigma^*)_k.\ P(w)$.

Another possibility: do induction on the length of w. Using this proof method, the statement to be shown is:

$$\forall n \in \mathbb{N}.\ \forall w \in \Sigma^*.\ (|w| = n) \text{ implies } P(w)$$

Singleton Languages Detail (2)

The proof proceeds by induction on word length; the statement to be proved is $\forall n \in \mathbb{N}.\ Q(n)$, where $Q(n)$ is "$\forall w \in \Sigma^*.\ (|w| = n) \longrightarrow \{w\}$ "is regular".

Base case. We must show $Q(0)$, i.e. that for any word w, if $|w| = 0$, then $\{w\}$ is regular. So fix w and assume that $|w| = 0$. This implies that $w = \varepsilon$. But $\{\varepsilon\}$ is regular, since the regular expression $\underline{\varepsilon}$ is such that $\mathcal{L}(\underline{\varepsilon}) = \{\varepsilon\}$.

Induction step. We must show that for any n, $Q(n) \longrightarrow Q(n+1)$. So fix n and assume (induction hypothesis) that $Q(n)$ holds. We must prove $Q(n+1)$, i.e. that for any w of length $n+1$, $\{w\}$ is regular. Now fix w and assume that $|w| = n+1$; we must find a regular expression $\underline{r}$ such that $\mathcal{L}(\underline{r}) = \{w\}$.

Singleton Languages Detail (3)

By definition of $|w|$, since $|w| = n+1$ there must exist $a \in \Sigma$ and $w' \in \Sigma^*$ such that $w = a \circ w'$ and $|w'| = n$. The induction hypothesis guarantees that $\{w'\}$ is regular, i.e. that there is a regular expression $\underline{r'}$ with $\mathcal{L}(\underline{r'}) = \{w'\}$. Now consider the regular expression $\underline{r} = \underline{a \circ r'}$.

$$\begin{array}{rcll} \mathcal{L}(\underline{r}) & = & \mathcal{L}(\underline{a \circ r'}) & \text{Definition of } \underline{r} \\ & = & \{a\} \circ \{w'\} & \text{Definition of } \mathcal{L} \\ & = & \{a \circ w'\} & \text{Definition of } \circ \\ & = & \{w\} & \end{array}$$

Consequently, $\{w\}$ is regular.

Closure Properties for Regular Languages

$\boxed{\text{Theorem}}$ The class of regular languages is closed with respect to $\cup$, $\circ$, and *.

For example, consider language union.

Suppose that L_1 and L_2 are regular; we want to prove that $L_1 \cup L_2$ is also regular. To do so, we must find a regular expression $\underline{r}_{12}$ such that $\mathcal{L}(\underline{r}_{12}) = L_1 \cup L_2$.

Since L_1 and L_2 are regular there exist regular expressions $\underline{r}_1, \underline{r}_2$ such that $\mathcal{L}(\underline{r}_1) = L_1$ and $\mathcal{L}(\underline{r}_2) = L_2$. Now consider $\underline{r}_{12} = \underline{r_1 + r_2}$.

$$\begin{array}{rcll} \mathcal{L}(\underline{r}_{12}) & = & \mathcal{L}(\underline{r_1 \cup r_2}) & \text{Definition of } \underline{r}_{12} \\ & = & \mathcal{L}(\underline{r}_1) \cup \mathcal{L}(\underline{r}_2) & \text{Definition of } \mathcal{L} \\ & = & L_1 \cup L_2 & \text{Assumption} \end{array}$$

Conseqently, $L_1 \cup L_2$ is regular.

Kleene's Theorem

Relating Automata and Regular Languages

So far we have three ways of “defining” languages:

- Regular expressions
- DFAs
- NFAs

We also know that that languages definable using DFAs are the same as those definable using NFAs.

What about languages definable using regular expressions?

They coincide with those for DFAs/NFAs!

Kleene’s Theorem

Theorem $L \subseteq \Sigma^*$ is regular if and only if there is a DFA M with $L = \mathcal{L}(M)$.

How can we show this? By giving constructions for converting:

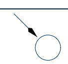
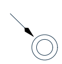
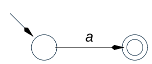
- regular expressions to DFAs; and
- DFAs to regular expressions.

Today we will only prove the first part.

Instead of building DFAs from regular expressions we will construct NFAs. (Why is this sufficient?)

Converting Regular Expressions into NFAs

Somehow, we need to get “operational content” (i.e. states and transitions) out of regular expressions. Basic regular expressions are easy:

Regular Expression	NFA
$\emptyset$	
ε	
$a (\in \Sigma)$	

But how do we handle the operators $\cup$, $\circ$ and $*$?

- Book uses on approach based on NFAs that also have ε-transitions.
- We’ll pursue a different approach.

Converting Regular Expressions into NFAs (cont.)

(Recall: $\mathcal{R}(\Sigma)$ is the set of regular expressions over Σ.)

1. We’ll define a predicate $\surd$ on regular expressions; $r\surd$ should hold exactly when $\varepsilon \in \mathcal{L}(r)$.
2. We’ll also define a relation $\longrightarrow \subseteq \mathcal{R}(\Sigma) \times \Sigma \times \mathcal{R}(\Sigma)$. Intuitively, $\longrightarrow$ should explain how to “build” words in $\mathcal{L}(r)$: if $r \xrightarrow{a} r'$ then any word $w' \in \mathcal{L}(r')$ should give rise to a word in $aw' \in \mathcal{L}(r)$.
3. We’ll then use these to construct a NFA from r as follows.
   - States are regular expressions.
   - Start state is r.
   - Transitions given by $\longrightarrow$
   - Accepting states given by $\surd$.

Defining $\surd$

Definition is recursive on structure of regular expressions!

Definition Let Σ be an alphabet. Then $\surd$ is defined as follows.

$$\underline{\varepsilon}\surd \text{ always} \quad (1)$$

$$\underline{r*}\surd \text{ always} \quad (2)$$

$$\underline{(r+s)}\surd \text{ if } \underline{r}\surd \quad (3)$$

$$\underline{(r+s)}\surd \text{ if } \underline{s}\surd \quad (4)$$

$$\underline{(rs)}\surd \text{ if } \underline{r}\surd \text{ and } \underline{s}\surd \quad (5)$$

Examples of $\surd$

$\underline{\varepsilon a*}\surd$	since $\underline{\varepsilon}\surd$ and $\underline{a*}\surd$
$\neg(\underline{(a+b)}\surd)$	since neither $\underline{a}\surd$ nor $\underline{b}\surd$
$\underline{01+(1+01)*}\surd$	since $\underline{(1+01)*}\surd$
$\neg(\underline{01(1+01)*}\surd)$	since $\neg(\underline{01}\surd)$

Proving $\surd$ Is Correct

Lemma Let $\underline{r}$ be a regular expression. Then $\varepsilon \in \mathcal{L}(\underline{r})$ iff $\underline{r}\surd$.

Proof Outline The proof proceeds by induction on $\underline{r}$, where the induction hypothesis allows the assumption of the result for "smaller" $\underline{r'}$. One would then do a case analysis based on the structure of $\underline{r}$:

- $\underline{r} = \underline{\emptyset}$
- $\underline{r} = \underline{\varepsilon}$
- $\underline{r} = \underline{a}$
- $\underline{r} = \underline{r_1 + r_2}$
- $\underline{r} = \underline{r_1 \circ r_2}$
- $\underline{r} = \underline{r_1 *}$

Defining $\longrightarrow$

Definition Let Σ be an alphabet. Then for $\underline{r}, \underline{r'} \in Reg(\Sigma)$ and $a \in \Sigma, \underline{r} \xrightarrow{a} \underline{r'}$ is defined as follows.

$$\underline{a} \xrightarrow{a} \underline{\varepsilon} \quad \text{if } a \in \Sigma \quad (1)$$

$$\underline{r+s} \xrightarrow{a} \underline{r'} \quad \text{if } \underline{r} \xrightarrow{a} \underline{r'} \quad (2)$$

$$\underline{r+s} \xrightarrow{a} \underline{s'} \quad \text{if } \underline{s} \xrightarrow{a} \underline{s'} \quad (3)$$

$$\underline{rs} \xrightarrow{a} \underline{r's} \quad \text{if } \underline{r} \xrightarrow{a} \underline{r'} \quad (4)$$

$$\underline{rs} \xrightarrow{a} \underline{s'} \quad \text{if } \underline{s} \xrightarrow{a} \underline{s'} \text{ and } \underline{r}\surd \quad (5)$$

$$\underline{r*} \xrightarrow{a} \underline{r'(r*)} \quad \text{if } \underline{r} \xrightarrow{a} \underline{r'} \quad (6)$$

Examples of $\longrightarrow$

- $\underline{0+1} \xrightarrow{0} \underline{\varepsilon}$ Why?

  $\underline{0} \xrightarrow{0} \underline{\varepsilon}$ By rule for $\underline{0}$

  $\underline{0+1} \xrightarrow{0} \underline{\varepsilon}$ By first rule for $\underline{\cup}$

- $\underline{(abb+a)*} \xrightarrow{a} \underline{\varepsilon bb(abb+a)*}$ Why?

  $\underline{a} \xrightarrow{a} \underline{\varepsilon}$ By rule for $\underline{a}$

  $\underline{abb} \xrightarrow{a} \underline{\varepsilon bb}$ By first rule for $\underline{\circ}$

  $\underline{abb+a} \xrightarrow{a} \underline{\varepsilon bb}$ By first rule for $\underline{\cup}$

  $\underline{(abb+a)*} \xrightarrow{a} \underline{\varepsilon bb(abb+a)*}$ By rule for $\underline{*}$

Proving $\longrightarrow$ Correct

Lemma Let $\underline{r} \in Reg(\Sigma)$, $a \in \Sigma$, and $w' \in \Sigma^*$. Then:

$$aw' \in \mathcal{L}(\underline{r}) \quad \text{iff} \quad \exists \underline{r}' \in Reg(\Sigma).\ \underline{r} \xrightarrow{a} \underline{r}' \text{ and } w' \in \mathcal{L}(\underline{r}')$$

Note This lemma says two things about $\longrightarrow$.

- If $\underline{r} \xrightarrow{a} \underline{r}'$ and $w' \in \mathcal{L}(\underline{r}')$ then $aw' \in \mathcal{L}(\underline{r})$.
- If $aw' \in \mathcal{L}(\underline{r})$ for some $a \in \Sigma$ then there is some $\underline{r}'$ such that $\underline{r} \xrightarrow{a} \underline{r}'$ and $w' \in \mathcal{L}(\underline{r}')$.

In other words, the construction of every non-ε element in $\mathcal{L}(\underline{r})$ can be "initiated" using $\longrightarrow$!

Computing Outgoing Transitions

In building NFAs we will need to be able to compute the set of outgoing transitions from regular expression $\underline{r}$, i.e. the set $\{\langle a, \underline{r}'\rangle \mid \underline{r} \xrightarrow{a} \underline{r}'\}$.

How do we do it? Recursively!

- If $\underline{r}$ is $\underline{\emptyset}$ or $\underline{\varepsilon}$, it has no transitions: $\{\ \}$.
- If $\underline{r}$ is a, it has one transition: $\{\langle a, \underline{\varepsilon}\rangle\}$.
- Otherwise, recursively compute transitions of subexpressions of $\underline{r}$. Then use rules to convert transitions of subexpressions into transitions for $\underline{r}$.

Example: Computing Outgoing Transitions

What are transitions of $\underline{0+1}$?

- Compute transitions of $\underline{0}$: $\{\langle 0, \underline{\varepsilon}\rangle\}$
- Compute transitions of $\underline{1}$: $\{\langle 1, \underline{\varepsilon}\rangle\}$
- From above and rules for $\underline{\cup}$, transitions for $\underline{0+1}$ are $\{\langle 0, \underline{\varepsilon}\rangle, \langle 1, \underline{\varepsilon}\rangle\}$

What are transitions of $\underline{a*b*}$?

- Compute transitions of $\underline{a*}$.
  - Compute transitions of $\underline{a}$: $\{\langle a, \underline{\varepsilon}\rangle\}$.
  - From above and rule for $\underline{*}$, transitions for $\underline{a*}$ are $\{\langle a, \underline{\varepsilon a*}\rangle\}$.
- Compute transitions for $\underline{b*}$: they are $\{\langle b, \underline{\varepsilon b*}\rangle\}$.
- Since $\underline{a*}\surd$, both rules for $\underline{\circ}$ are applicable, and transitions for $\underline{a*b*}$ are $\{\langle a, \underline{\varepsilon a*b*}\rangle, \langle b, \underline{\varepsilon b*}\rangle\}$.

Building NFAs Using $\longrightarrow$ and $\surd$

Suppose that

$$\underline{r}_0 \xrightarrow{a_1} \underline{r}_1 \xrightarrow{a_2} \cdots \underline{r}_{n-1} \xrightarrow{a_n} \underline{r}_n$$

and $\underline{r}_n\surd$. Then the lemmas about $\surd$ and $\longrightarrow$ guarantee that $a_1 ... a_n \in \mathcal{L}(\underline{r}_0)$.

This suggests a way to build a NFA from a regular expression $\underline{r}$.

- States are regular expressions "reachable" from $\underline{r}$ by some number of $\longrightarrow$ steps.
- Start state is $\underline{r}$.
- Transitions given by $\longrightarrow$.
- Accepting states given by $\surd$.

Building NFAs: Implementation

One way to implement previous strategy: build states, transitions in NFA for $\underline{r}$ in demand-driven manner.

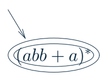
- Start with state set $Q = \{\underline{r}\}$.
- Maintain set $toProc$ of states whose outgoing transitions need to be calculated; initially, $toProc = \{\underline{r}\}$.
- While $toProc$ is nonempty, choose an element from it, compute outgoing transitions from it, and add target states of transitions to Q and $toProc$ if necessary.

Building NFA for $(abb \cup a)*$

Initially

$Q = \{\underline{(abb + a)*}\}$

$toProc = \{\underline{(abb + a)*}\}$

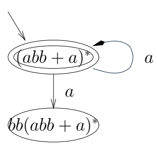


Transitions for $\underline{(abb + a)*}$:

$\{\langle a, \underline{(abb + a)*}\rangle, \langle a, \underline{bb(abb + a)*}\rangle\}$

$Q = \{\underline{(abb + a)*}, \underline{bb(abb + a)*}\}$

$toProc = \{\underline{bb(abb + a)*}\}$

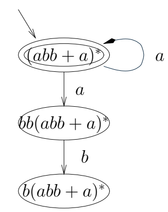


Transitions for $\underline{bb(abb + a)*}$:

$\{\langle b, \underline{b(abb + a)*}\rangle\}$

$Q = \{\underline{(abb + a)*}, \underline{bb(abb + a)*}, \underline{b(abb + a)*}\}$

$toProc = \{\underline{b(abb + a)*}\}$

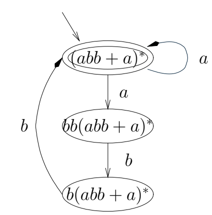


Transitions for $\underline{b(abb + a)*}$:

$\{\langle b, \underline{(abb + a)*}\rangle\}$

$Q = \{\underline{(abb + a)*}, \underline{bb(abb + a)*}, \underline{b(abb + a)*}\}$

$toProc = \emptyset$; we are finished!



Implications of Kleene's Theorem

1. Regular languages are closed with respect to complement and intersection.

2. Theorem has practical importance.

`ls *.c`	OS's convert regular expressions to DFAs to implement this
`egrep`	String search utility converts regular expressions to DFAs
`lex`	Scanner generator used in compiler construction; converts regular expressions for keywords, identifiers into DFAs