Welcome to

Cpts 317

Friday June 31

Topic of the day:

Regular Expressions
Regular Expressions

Much like we use operations $+$ and $\times$ in arithmetic to build expressions such as $(2 + 6) \times 3$,
we use regular operations such as union, concatenation and star to build regular expressions.

Example

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

The value of an arithmetic expression is a number.

The value of a regular expression is a language.
Regular expressions have an important role in computer science applications.

- Unix
- Programming languages such as Perl
- Text editor

provide mechanisms for description of patterns using regular expressions.
Precedence order in regular expressions:

1. Star
2. Concatenation
3. Union

Unless parentheses is used, which change the usual way.
Formal definition of regular expression

We say that \( R \) is a regular expression if \( R \) is:
1. \( a \) for some \( a \) in alphabet \( \Sigma \).
2. \( \varepsilon \)
3. \( \emptyset \)
4. \( (R_1 \cup R_2) \), where \( R_1 \) and \( R_2 \) are reg. exp.
5. \( (R_1 \cdot R_2) \), where \( R_1 \) and \( R_2 \) are reg. exp.
6. \( (R_1)^* \), where \( R_1 \) is a regular exp.

In item 1 and 2, the reg. expressions \( a \) and \( \varepsilon \) represent the language \( \{a\} \) and \( \{\varepsilon\} \), respectively.
In item 3, the reg. exp. \( \emptyset \) represents the empty language. In 4-6, the expressions represent the language obtained by taking union, concatenation, and Kleene star.
A few points

* We are defining regular expressions in terms of smaller regular expressions (and not circularly). \( \rightarrow \) inductive definition.

* \( R^+ \) is usually used as a shorthand for \( RR^* \)

* \( R^k \) is used as a shorthand for concatenation of \( k \) \( R \)'s

* When we want to distinguish between a regular expression \( R \) and the language it describes, we write \( L(R) \) to be the language of \( R \).
Example

1. $0^* 1^* 0^* = \{ w \mid w \text{ contains a single } 1 \}$

2. $1^* (01^*)^* = \{ w \mid w \text{ every zero is followed by at least one } 1 \}$

3. $\Sigma^* 001\Sigma^* = \{ w \mid w \text{ contains 001 or a substring } \}$

4. $\Sigma^* 1\Sigma^* = \{ w \mid w \text{ contains at least one } 1 \}$

5. $(\Sigma \Sigma)^* = \{ w \mid w \text{ is a string of even length} \}$

6. $(\Sigma \Sigma \Sigma)^* = \{ w \mid w \text{ is a string of any length that is a multiple of } 3 \}$

7. $\Sigma^* 0 \Sigma^* 1 \Sigma^* 1 \Sigma^* 0 \Sigma^* 1 = \{ w \mid w \text{ starts and ends with the same substring} \}$

8. $1^* \emptyset = \emptyset$

9. $\emptyset^* = \{ \emptyset \}$

10. $(0 \cup 1) (1 \cup 0)^* = \{ 3, 0, 1, 01 \}$
Identitites

Let $R$ be any regular expression,

* $R \cup \emptyset = R$
* $R \cdot \varepsilon = R$

However

* $R \cup \varepsilon$ may not equal $R$
  
  e.g. if $R = 0$, then $L(R) = \{0\} \quad \text{but} \quad L(R \cup \varepsilon) = \{0, 3\}$

* $R \cdot \emptyset$ may not equal $R$
  
  e.g. if $R = 0$, then $L(R) = \{0\} \quad \text{but} \quad L(R \cdot \emptyset) = \emptyset$. 
Equivalence with Finite Automata

**Theorem**

A language is regular if and only if some regular expression describes it.

Somewhat surprisingly, some finite automata and regular expressions are outwardly different:

finite automata = machine

regular expression = description

We will break the proof of this theorem into its two directions.
Lemma

If a language is described by a regular expression, then it is regular.

Proof idea

Say we have a regular expression \( R \) describing some language \( A \).

We will show how to convert \( R \) into an NFA recognizing \( A \).

By an earlier result we saw, if an NFA recognizes \( A \) then \( A \) is regular.
Proof

Let us convert $R$ into an NFA $N$. We consider the six cases in the formal definition of regular expressions.

1. $R = a$ for some $a \in \Sigma$.
   Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.

2. $R = \epsilon$. Then $L(R) = \{\epsilon\}$, and the following NFA recognizes $L(R)$.

3. $R = \phi$.
   Then $L(R) = \phi$, and the following NFA recognizes $L(R)$.

\[
N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})
\]

\[
\delta(q_1, a) = \{q_2\}
\]

\[
\delta(q_1, b) = \phi \text{ for } q_1 \neq q_1
\]

\[
\delta(q_2, b) = \phi \text{ for } q_2 \neq q_1
\]
4. \( R = R_1 \cup R_2 \)

5. \( R = R_1 \circ R_2 \)

6. \( R = R_1^* \)

For 4-6, we use the construction.

In the proof, we saw in the last two lectures.

Before seeing the other direction of the theorem, let us make use of this proof to see some examples.
Example

Convert the regular expression

\((ab \cup a)^*\)

to an NFA.
Lemma

If a language is regular, then it is described by a regular expression.

Proof IDEA

Need to show that if a language $A$ is regular, a regular expression describes it.

Because $A$ is regular, it is accepted by a DFA. We describe a procedure for converting DFAs into equivalent regular expressions.

We break this procedure into two parts:

1. DFA $\rightarrow$ Generalized NFA (GENFA)
2. GENFA$^*$ $\rightarrow$ regular expression
GNFAs

* are NFA, wherein transition arrows may have any regular expressions as labels, instead of any member of the alphabet or ε.

* GNFA reads blocks of symbols from the input, not necessarily one symbol at a time as in ordinary NFA.

* The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow.
GNFA could

A GNFA is nondeterministic and so may have several different ways to process the same input string.

It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input.

Example

\[
\begin{align*}
&G_{start} \quad ab^* \quad aa \\
&\quad \quad \quad a^* \quad (aa)^* \\
&\quad \quad \quad ab \quad b^* \\
&\quad \quad \quad ab \cup ba \\
&\quad \quad \quad b \\
&\quad \quad \quad gb \\
&G_{accept}
\end{align*}
\]
For convenience, we require that GNFAs always have a special form that meets the following conditions.

1. The start state has transition arrows going to every other state but no arrows coming in from any other state.

2. There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.

3. Except for the start & accept states, one arrow goes from every state to every other state and also from each state itself.
To Convert a DFA into a GNFA

In the Special form

* We add a new start state with an E arrow to the old start state, and a new accept state with E arrows from the old accept state.

* If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels.

* Finally, we add arrows labelled $\emptyset$ between states that had no arrows.
* Now we show how to convert a GNFA into a regular expression.

* Suppose the GNFA has \( k \) states.

Then, because a GNFA must have a start and an accept state and they must be different from each other, we know that \( k \geq 2 \).

* If \( k > 2 \), we construct an equivalent GNFA with \( k - 1 \) states. This step can be repeated on the new GNFA until it is reduced to two states.
If $k = 2$, the GNFA has a single arrow that goes from the start state to the accept state. The label of this arrow is the equivalent regular expression.

For example, the stages in constructing a GNFA with three states to an equivalent regular expression are shown in the following figure.
The crucial step is constructing an equivalent GNFA with one fewer state when \( k > 2 \).

We do so by selecting a state, rippin it out of the machine, and repairing the remainder so that the same language is still recognized.

Any state will do, provided that it is not the start state or accept state.

We are guaranteed that such a state will exist because \( k > 2 \). Let us call the removed state \( f \).
* After removing $\ell_0$, we repair the machine by altering the regular expressions that each of the remaining arrows.

* The new labels compensate for the absence of $\ell_0$ by adding them in the cost computations.

* The new label going from a state $q_i$ to a state $q_j$ is a regular expression that describes all strings that would take the machine from $q_i$ to $q_j$ either directly or via $\ell_0$. 
Example

In the old machine, if

1. \( f_i \) goes to \( \text{Trip} \) with an arrow labeled \( R_1 \),

2. \( \text{Trip} \) goes to itself with an arrow labeled \( R_2 \),

3. \( \text{Trip} \) goes to \( s_j \) with an arrow labeled \( R_3 \), and

4. \( f_i \) goes to \( f_j \) with an arrow labeled \( R_4 \).