Regular Expressions

- If R and S are regular expressions, so are:
  - ε: empty string
  - a: for any character a
  - RS: (concatenation: "R followed by S")
  - R | S: (alternation: "R or S")
  - R*: (Kleene star: "zero or more R's")

Regular Expression Extensions

- If R is a regular expressions, so are:
  - R?: = ε | R (zero or one R)
  - R+ = RR* (one or more R’s)
  - (R) = R (no effect: grouping)
  - [abc] = a|b|c (any of the listed)
  - [a-e] = a|b|...| e (character ranges)
  - [^ab] = c|d|... (anything but the listed chars)

Concepts

- Tokens = strings of characters representing the lexical units of the programs, such as identifiers, numbers, keywords, operators
  - May represent a unique character string (keywords, operators)
  - May represent multiple strings (identifiers, numbers)

- Regular expressions = concise description of tokens
  - A regular expressions describes a set of strings

- Language denoted by a regular expression = the set of strings that it represents
  - L(R) is the language denoted by regular expression R

How To Use Regular Expressions

- We need a mechanism to determine if an input string w belongs to the language denoted by a regular expression R

  ![Diagram](https://via.placeholder.com/150)

  Input string w in the program

  Regex R which describes a token

  Yes, if w is token
  No, if w is not token

- Such a mechanism is called an acceptor
**Acceptors**
- **Acceptor** = determines if an input string belongs to a language \( L \)

\[
\begin{array}{c|c|c}
\text{Input String} & \text{Acceptor} & \\
\hline
w & \{ \text{Yes, if } w \in L \} & \{ \text{No, if } w \notin L \}
\end{array}
\]

- **Finite Automata** = acceptor for languages described by regular expressions

**Finite Automata**
- Informally, finite automata consist of:
  - A finite set of states
  - Transitions between states
  - An initial state (start state)
  - A set of final states (accepting state)
- Two kinds of finite automata:
  - Deterministic finite automata (DFA): the transition from each state is uniquely determined by the current input character
  - Non-deterministic finite automata (NFA): there may be multiple possible choices or some transitions do not depend on the input character

**DFA Example**
- Finite automaton that accepts the strings in the language denoted by the regular expression \( ab^*a \)

\[
\begin{array}{c|c|c}
\text{transition table} & a & b \\
\hline
0 & 1 & Error \\
1 & 2 & 1 \\
2 & Error & Error
\end{array}
\]

**Simulating the DFA**
- Determine if the DFA accepts an input string

\[
\text{trans\_table}[\text{NSTATES}][\text{NCHARS}]
\]
\[
\text{accept\_states}[\text{NSTATES}]
\]
\[
\text{state} = \text{INITIAL}
\]
\[
\text{while} (\text{state} \neq \text{ERROR}) \\
\quad \text{c} = \text{input.read}(); \\
\quad \text{if} (\text{c} \neq \text{EOF}) \text{break}; \\
\quad \text{state} = \text{trans\_table}[	ext{state}][\text{c}]; \\
\text{return} \text{accept\_states}[	ext{state}];
\]

**RE → Finite automaton?**
- Can we build a finite automaton for every regular expression?
- Strategy: build the finite automaton inductively, based on the definition of regular expressions

\[
\varepsilon \quad a
\]

**RE → Finite automaton?**
- Alternation \( R | S \)

\[
\oplus
\]

- Concatenation: \( RS \)

\[
\rightarrow
\]
NFA Definition
- A non-deterministic finite automaton (NFA) is an automaton where the state transitions are such that:
  - There may be e-transitions (transitions which do not consume input characters)
  - There may be multiple transitions from the same state on the same input character

Example: regexp?

RE ⇒ NFA intuition

NFA construction
- NFA only needs one stop state (why?)
- Canonical NFA:

Use this canonical form to inductively construct NFAs for regular expressions

Inductive NFA Construction

DFA vs NFA
- DFA: action of automaton on each input symbol is fully determined
  - obvious table-driven implementation
- NFA:
  - automaton may have choice on each step
  - automaton accepts a string if there is any way to make choices to arrive at accepting state / every path from start state to an accept state is a string accepted by automaton
  - not obvious how to implement!

Simulating an NFA
- Problem: how to execute NFA?
  "strings accepted are those for which there is some corresponding path from start state to an accept state"
- Conclusion: search all paths in graph consistent with the string
- Idea: search paths in parallel
  - Keep track of subset of NFA states that search could be in after seeing string prefix
  - "Multiple fingers" pointing to graph
Example

- Input string: -23
- NFA states:
  - \{0, 1\}
  - \{1\}
  - \{2, 3\}
  - \{2, 3\}

NFA-DFA conversion

- Can convert NFA directly to DFA by same approach
- Create one DFA for each distinct subset of NFA states that could arise
- States: \{(0, 1), \{1\}, \{2, 3\}\}

Algorithm

- For a set \(S\) of states in the NFA, compute \(\epsilon\)-closure(S) = set of states reachable from states in \(S\) by \(\epsilon\)-transitions
  - \(T = S\)
  - Repeat \(T = T \cup \{s | s \in T, (s,s') is \epsilon\)-transition\)
  - Until \(T\) remains unchanged
  - \(\epsilon\)-closure(S) = \(T\)
- For a set \(S\) of states in the NFA, compute \(\text{DFAEdge}(S,c) = \) the set of states reachable from states in \(S\) by transitions on character \(c\) and \(\epsilon\)-transitions
  - \(\text{DFAEdge}(S,c) = \epsilon\)-closure( \(\{ s | s \in S, (s,s') is \epsilon\)-transition\) )

Algorithm

\[
\text{DFA-initial-state} = \epsilon\text{-closure}(\text{NFA-initial-state})
\]
\[
\text{Worklist} = (\text{DFA-initial-state})
\]
\[
\text{While ( Worklist not empty )}
\]
\[
\text{Pick state \(S\) from Worklist}
\]
\[
\text{For each character } c
\]
\[
S' = \text{DFAEdge}(S,c)
\]
\[
\text{if ( } S' \text{ not in DFA states)}
\]
\[
\text{Add } S' \text{ to DFA states and worklist}
\]
\[
\text{Add an edge ( } S, S' \text{) labeled } c \text{ in DFA}
\]
\[
\text{For each DFA-state } S
\]
\[
\text{If } S \text{ contains an NFA-final state}
\]
\[
\text{Mark } S \text{ as DFA-final-state}
\]

Putting the Pieces Together

Regular Expression \(R\) \(\Rightarrow\) NFA Conversion

Input String \(w\) \(\Rightarrow\) DFA Simulation

Yes, if \(w \in L(R)\)
No, if \(w \notin L(R)\)