Outline

- Regexpr review
- DFAs, NFAs
- DFA simulation
- RE-NFA conversion
- NFA-DFA conversion

Regular Expressions

- If $R$ and $S$ are regular expressions, so are:
  - $\emptyset$ 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:
  - $R^+$ = $R | R$ (zero or one $R$)
  - $R^*$ = $R^* | R$ (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)
  - $[^ a b]$ = $c | d | ...$ (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 expression 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 $R$
  - Input string $w$ in the program
  - Regex $R$ which describes a token
  - Yes, if $w$ = token
  - No, if $w$ $\neq$ token

- Such a mechanism is called an acceptor
### Acceptors

- **Acceptor** = determines if an input string belongs to a language $L$

\[
\text{Input String } \rightarrow \text{Acceptor} \rightarrow \begin{cases} \text{Yes, if } w \in L \\ \text{No, if } w \notin L \end{cases}
\]

- **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 $ab^*a$

  - A graph
    \[
    \begin{array}{c}
    0 \\
    a \\
    1 \\
    a \\
    2 \\
    \end{array}
    \]

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

### Simulating the DFA

- Determine if the DFA accepts an input string

```java
trans_table[NSTATES][NCHARS]
accept_states[NSTATES]
state = INITIAL

while (state != ERROR) {
    c = input.read();
    if (c == EOF) break;
    state = trans_table[state][c];
}
return accept_states[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

  - $\epsilon$
    \[
    \begin{array}{c}
    \epsilon \\
    \end{array}
    \]

  - $a$
    \[
    \begin{array}{c}
    a \\
    \end{array}
    \]

  - $R \cup S$
    \[
    \begin{array}{c}
    R \cup S \\
    \end{array}
    \]

  - $R \cdot S$
    \[
    \begin{array}{c}
    R \cdot S \\
    \end{array}
    \]
NFA Definition

- A non-deterministic finite automaton (NFA) is an automaton where the state transitions are such that:
  - There may be ε-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 \( \Rightarrow \) NFA intuition

```
- \([0-9]+\)  \((-[0-9]+[0-9]+\)?
```

NFA construction

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

```
use this canonical form to inductively construct NFAs for regular expressions
```

Inductive NFA Construction

```
R S
```

```
R | S
```

```
R*
```

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 string could be in after seeing string prefix
  - “Multiple fingers” pointing to graph
Example

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

![Diagram of NFA states with transitions]

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\}

![Diagram of DFA states with transitions]

Algorithm

- For a set \(S\) of states in the NFA, compute 
  \(\epsilon\)-closure\((S)\) = the set of states reachable from states in 
  \(S\) by \(\epsilon\)-transitions

\[
T = S \\
\text{Repeat } T = T \cup \{s \mid \exists t \in T \text{ such that } (s, t, \epsilon) \in \text{transition}\} \\
\text{until } \text{T remains unchanged} \\
\epsilon\text{-closure}(S) = T
\]

- For a set \(S\) of states in the NFA, compute 
  \(\text{DFA-\text{edge}}(S, c)\) = the set of states reachable from states in 
  \(S\) by transitions on character \(c\) and \(\epsilon\)-transitions

\[
\text{DFA-\text{edge}}(S, c) = \epsilon\text{-closure}\{s \mid s \in S, (s', s) \text{ is \epsilon-transition}\}
\]

Algorithm

- Top-level algorithm:
  \(\text{DFA-initial-state} = \epsilon\text{-closure}(\text{NFA-initial-state})\)

  For each \(\text{DFA-state } S\)
  
  For each character \(c\)
  
  \(S' = \text{DFA-\text{edge}}(S, c)\)

  Add an edge \((S, S')\) labeled with
  character \(c\) in the DFA

  For each \(\text{DFA-state } S\)
  
  if \(S\) contains an \(\text{NFA-final-state}\)
  
  Mark \(S\) as \(\text{DFA-final-state}\)

Putting the Pieces Together

Regular
Expression \(R\)

\(\text{RE} \Rightarrow \text{NFA Conversion}\)

\(\text{NFA} \Rightarrow \text{DFA Conversion}\)

\(\text{DFA Simulation}\)

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