Compiler Development (CMPSC 401)

Lexical Analysis

Janyl Jumadinova

January 29, 2019
Deterministic Finite Automata (DFAs), Non-deterministic Finite Automata (NFAs) and REs have same expressive power i.e. allow precisely same patterns/sets to be specified.

For every DFA there is an equivalent RE

For every NFA there is an equivalent DFA

For every RE there is an equivalent NFA
Finite State Automaton

A finite automaton is a machine that has a finite number of states and a finite number of transitions between these.

- One marked as initial state.
- One or more marked as final states.
- States sometimes labeled or numbered.
A finite automaton is a machine that has a finite number of states and a finite number of transitions between these.

- One marked as initial state.
- One or more marked as final states.
- States sometimes labeled or numbered.

A set of transitions from state to state.

- Each labeled with symbol from $\sum$ (the alphabet), or $\varepsilon$.
- The symbols correspond to characters in the input string.
Example

![Diagram of a state machine with transitions labeled 0 and 1. The diagram starts with a round box labeled "start" and transitions to two paths, one labeled 0 and 1, leading to a loop.]
Notice that there are multiple transitions defined here on 0 and 1. If we read a 0 or 1 here, we follow both transitions and enter multiple states.
Example
Example

![Example Diagram]

- **Start State**: Yellow circle
- **States**: Multiple states with transitions labeled 0, 1
- **Transitions**: Arrows connecting states
- **Input Sequence**: 0 1 1 1 1 0 1
Example

```
0 1 1 1 0 1
```

```
start

0 0, 1
1
```

```
0 1
0
1
```
Example
Example
Example
Example
Example

$\begin{array}{c}
0 \\
1 \\
0, 1
\end{array}$

$\begin{array}{c}
\text{start} \\
0, 1
\end{array}$

$\begin{array}{c}
0 \\
1
\end{array}$

$\begin{array}{c}
1 \\
0
\end{array}$

$\begin{array}{c}
1
\end{array}$
Example

```
0 1 1 1 0 1
```

```
0 → 0, 1
1 → 0
```

```
0 → 0
1 → 1
```
Example

![Diagram of a finite state machine with transitions labeled 0, 1. The diagram shows states connected with arrows indicating the transition on input 0 or 1.]

0 1 1 1 1 0 1
Example

Diagram of a finite state machine with transitions labeled 0, 1, and 0, 1, respectively. The sequence 0 1 1 1 0 1 is indicated as input to the machine.
Example

Since we are in at least one accepting state, the automaton accepts.
Finite State Automaton

- Operate by reading input symbols (usually characters).
  - Transition can be taken if labeled with current symbol.
  - $\varepsilon$-transition can be taken at any time.

Accept when final state reached and no more input.

Slightly different in a scanner, where the FSA is used as a subroutine to find the longest input string that matches a token RE.

Reject if no transition possible, or no more input and not in final state.
Finite State Automaton

- Operate by reading input symbols (usually characters).
  - Transition can be taken if labeled with current symbol.
  - $\varepsilon$-transition can be taken at any time.
- Accept when final state reached and no more input.
Finite State Automaton

- Operate by reading input symbols (usually characters).
  - Transition can be taken if labeled with current symbol.
  - \( \varepsilon \)-transition can be taken at any time.
- Accept when final state reached and no more input.
  - Slightly different in a scanner, where the FSA is used as a subroutine to find the longest input string that matches a token RE.
Finite State Automaton

- Operate by reading input symbols (usually characters).
  - Transition can be taken if labeled with current symbol.
  - $\varepsilon$-transition can be taken at any time.
- Accept when final state reached and no more input.
  - Slightly different in a scanner, where the FSA is used as a subroutine to find the longest input string that matches a token RE.
- Reject if no transition possible, or no more input and not in final state.
A More Complex Automaton
A More Complex Automaton

These are called \textbf{\textit{\( \epsilon \)-transitions}}. These transitions are followed automatically and without consuming any input.
A More Complex Automaton

![Automaton Diagram]
A More Complex Automaton

\[ \text{Diagram of automaton with states and transitions.} \]

Start state labeled "\( \varepsilon \)" transitions to state labeled "a, b".

State labeled "a, b" transitions to state labeled "c".

State labeled "c" transitions back to state labeled "a, b".

State labeled "b, c" transitions to state labeled "a, b".

State labeled "a, c" transitions to state labeled "b, c".
A More Complex Automaton
A More Complex Automaton
A More Complex Automaton

```
start

ε → a, c → b, c

ε → a, b

- c

ε

b c b a
```
A More Complex Automaton
A More Complex Automaton

Diagram:

- Start state:
  - Transitions:
    - ε to ε
    - ε to b, c
- State 1:
  - Transitions:
    - ε to ε
    - a, b to ε
  - Final state
  - Transitions:
    - ε to b, c
- State 2:
  - Transitions:
    - c to c
    - b to b
- State 3:
  - Transitions:
    - a to a

Example string: b c b a
A More Complex Automaton
A More Complex Automaton
A More Complex Automaton
A More Complex Automaton
DFA vs. NFA

- **Deterministic Finite Automata (DFA)**
  - No choice of which transition to make.
DFA vs. NFA

- **Deterministic Finite Automata (DFA)**
  - No choice of which transition to make.

- **Non-deterministic Finite Automata (NFA)**
  - Choice of transition in at least one case.
DFA vs. NFA

- **Deterministic Finite Automata (DFA)**
  - No choice of which transition to make.

- **Non-deterministic Finite Automata (NFA)**
  - Choice of transition in at least one case.
  - \( \varepsilon \) transitions (arcs): If the current state has any outgoing \( \varepsilon \) arcs, we can follow any of them without consuming any input.
  - Modeling choice option 1: guess path, backtrack if rejects
    Option 2: “clone” at choice point, accept if any clone accepts.
Simulating an NFA

For each character in the input:
  - For each current state:
    - Follow all transitions labeled with the current letter.
    - Add these states to the set of new states.

Add every state reachable by an ε-move to the set of next states.
Accept if some way to reach a final state on given input.
Reject if no possible way to final state.
Simulating an NFA

For each character in the input:
  - For each current state:
    - Follow all transitions labeled with the current letter.
    - Add these states to the set of new states.
  - Add every state reachable by an \( \varepsilon \)-move to the set of next states.

Accept if some way to reach a final state on given input.
Reject if no possible way to final state.
Simulating an NFA

- For each character in the input:
  - For each current state:
    - Follow all transitions labeled with the current letter.
    - Add these states to the set of new states.
  - Add every state reachable by an \( \varepsilon \)-move to the set of next states.

- Accept if some way to reach a final state on given input.

- Reject if no possible way to final state.
Want DFA for speed (no backtracking or cloning).
But conversion from regular expressions to NFA is easier.
Luckily, there is a well-defined procedure for converting an NFA to an equivalent DFA.
Usefulness of RE to NFA Construction

**Lexical Analysis**
- Specify language tokens (identifiers, numerical constants, symbols etc.) as REs.
- Tools like `lex` automatically generate automaton-based code to decompose source code into constituent tokens.

**Pattern Matching** e.g. text editors, `grep`
- Pattern specified as RE.
- Automaton-based search locates occurrences.
Lexical Analysis Generators

- Generates analyzer automatically from “descriptions” (regular expressions/ NFAs) of tokens in the programming language.
- Examples:
  - lex/flex for C
  - jFlex for Java
Terminology

- **A token** is a group of characters having collective meaning.
- **A lexeme** is an actual character sequence forming a specific instance of a token, such as `num`.
- **A pattern** is a rule expressed as a regular expression and describing how a particular token can be formed. For example, `^[A-Za-z] [A-Za-z 0-9]*` is a rule.
(jF)Lex

- **Input:**
  - description of token structure (regular expressions)
  - information on how to “process” different tokens
Input:
- description of token structure (regular expressions)
- information on how to “process” different tokens

Output: an implementation of NFA-based function that:
- recognizes tokens (as specified by RE rules)
- processes them (as specified by actions)
jFlex Program Format

/* User code */
%
/* Options and declarations */
%
/* Lexical Rules */

1 User Code (e.g. import statements), included top of generated Java; often empty.

2 Options “Marcos” (named REs); code to be spliced into generated Java class.

3 Rule = Pattern + Action.

4 Pattern = Regular Expression.

5 Action = Snippet of Java code (Actions triggered whenever pattern matched).
## jFlex RE Syntax

<table>
<thead>
<tr>
<th>pattern</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td>character a</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>character a (even special chars.)</td>
</tr>
<tr>
<td><code>abc</code></td>
<td>a followed by b followed by c (no explicit concat. symbol)</td>
</tr>
<tr>
<td>`a</td>
<td>b`</td>
</tr>
<tr>
<td><code>a*</code></td>
<td>zero or more rep. of a</td>
</tr>
<tr>
<td><code>a+</code></td>
<td>one or more rep. of a</td>
</tr>
<tr>
<td><code>a?</code></td>
<td>optional a</td>
</tr>
<tr>
<td><code>(a)</code></td>
<td>a itself</td>
</tr>
<tr>
<td><code>[abc]</code></td>
<td>any (one) of a or b or c</td>
</tr>
<tr>
<td><code>[^abc]</code></td>
<td>any char. except a, b or c</td>
</tr>
<tr>
<td><code>.</code></td>
<td>any char. except newline</td>
</tr>
</tbody>
</table>