CSE450
Translation of Programming Languages
Automata, Simple Language Design Principles
Finite Automata State Graphs

A state:

The start state:

An accepting state:

A transition:
A Simple Example

A finite automaton that accepts only "1":

![Finite Automaton Diagram]
Another Simple Example

Alphabet: \{0, 1\}

A finite automaton accepting any number of 1's followed by a single 0.
And Another Example

Alphabet: \{0, 1\}

What language does this recognize?
Another Simple Example

Alphabet still \{0, 1\}

The operation of the automaton is not completely defined by the input
- On input "11" the automaton could be in either state
Epsilon Moves

Another kind of transition: \( \varepsilon \)-moves
Deterministic and Non-deterministic Automata

- **Deterministic Finite Automata (DFA)**
  - One transition per input per state
  - No \( \varepsilon \)-moves

- **Non-deterministic Finite Automata (NFA)**
  - Can have multiple transitions for one input in a given state
  - Can have \( \varepsilon \)-moves

- **Finite automata have finite memory**
  - Need only to encode the current state
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by the input

- NFSs can choose
  - Whether to make $\varepsilon$-moves
  - Which of multiple transitions for a single input to take
Acceptance of NFAs

An NFA can get into multiple states

Input: 100
Rule: NFA accepts if it can get in a final state.
NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)
- NFAs are easier to design
  - You can just list all the rules you want to allow
- DFAs are easier to implement
  - There are no choices to consider
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

- DFA can be exponentially larger than the NFA
Lexical Analysis Overview

- Regular Expressions
- Lexical Specification
- Non-deterministic Finite Automata
- Deterministic Finite Automata
- Table-driven Implementation of DFA
For each kind of regular expression, we can define an NFA. We will do this by using recursive definition.

Notation: NFA for regular expression $A$:

**Base cases:**

For $\varepsilon$

For input 'a'
Regular Expressions to NFA (2)

For $AB$

For $A | B$
Regular Expressions to NFA (3)

For A*

How do we do "A+"?

What about "A?"?
Example of RE to NFA Conversion

Consider the regular expression \((1 \mid 0)^* 1\)

The NFA is:
Lexical Analysis Overview

Regular Expressions

Lexical Specification

Non-deterministic Finite Automata

Deterministic Finite Automata

Table-driven Implementation of DFA
NFA to DFA: The Trick

- Simulate the NFA
- Each state of DFA
  = a non-empty subset of states of the NFA
- Start state
  = the set of NFA states reachable through ε-moves from the NFA start state
- Add a transition \((S, a) \rightarrow S'\) to DFA if and only if:
  - \(S'\) is the set of NFA states reachable from the states in \(S\) after seeing the input \(a\)
    (remember to consider ε-moves as well!)
NFA to DFA Example
NFA to DFA Remark

- An NFA may be in many states at any time.
- How many different possible states are there?
- If there are \( n \) states, the NFA must be in some subset of those \( n \) states.
- How many non-empty subsets are there?
  - \( 2^n - 1 = \text{finitely many} \)
Lexical Analysis Overview

- Regular Expressions
- Lexical Specification
- Non-deterministic Finite Automata
- Deterministic Finite Automata
- Table-driven Implementation of DFA
Table Implementation

- A DFA can be implemented by a 2-dimensional table $T$
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $(S_i, a) \rightarrow S_k$ define $T[i, a] = k$

- DFA "execution"
  - If in state $S_i$ and input $a$, read $T[i, a] = k$ and go to state $S_k$
  - Very efficient
Table Implementation of DFA

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>
Final notes about Implementation

- NFA to DFA conversion is at the heart of tools such as Lex
- But, DFAs can be huge
- In practice, lex-like tools trade off speed for space in the choice of NFA and DFA representations
Simple Language Design Principles
Notes on Language Design

In 1986, Jon Bentley wrote a Programming Pearls column in Communications of the ACM about "Little Languages". In it, he lists seven criteria for designing a little language.

1. **Orthogonality**: Different statements in the language should each have their own purpose.

2. **Generality**: Each command in the language should be as versatile as possible as long as it does not make it much more complex.
3. **Parsimony**: Do not use more commands in your language that it really needs to have.

4. **Completeness**: Make sure your language can do everything you need it to.

5. **Similarity**: Use language components that the user will understand.

6. **Extensibility**: Your language should be easy to add new commands and features to.

7. **Openness**: You should be able to interface your language with other available languages.