CS345H: Programming Languages

Lecture 4: Implementation of Lexical Analysis

Thomas Dillig
Announcements

- WA1 and PA0 are due Today
Announcements

- WA1 and PA0 are due Today
- WA2 and PA1 out today :-)

Thomas Dillig, CS345H: Programming Languages  Lecture 4: Implementation of Lexical Analysis 2/33
Announcements

- WA1 and PA0 are due Today
- WA2 and PA1 out today :-) 
- If you are not very, very busy right now, get started now
Outline

- Last time: Specifying lexical structure using regular expressions
Last time: Specifying lexical structure using regular expressions

Today: How to recognize strings matching regular expressions using finite automata.
Outline

- Last time: Specifying lexical structure using regular expressions
- Today: How to recognize strings matching regular expressions using finite automata.
- We will see determinist finite automata (DFAs) and non-deterministic finite automata (NFAs)
Last time: Specifying lexical structure using regular expressions

Today: How to recognize strings matching regular expressions using finite automata.

We will see determinist finite automata (DFAs) and non-deterministic finite automata (NFAs)

High-level story: RegEx -> NFA -> DFA -> Tables
Regular Expressions in Lexical Specifications

▸ Last lecture: How to specify the predicate $s \in L(R)$
Regular Expressions in Lexical Specifications

- Last lecture: How to specify the predicate \( s \in L(R) \)
- But yes/no answer is not enough!
Regular Expressions in Lexical Specifications

- Last lecture: How to specify the predicate $s \in L(R)$
- But yes/no answer is not enough!
- We really want to partition input into tokens
Regular Expressions in Lexical Specifications

- Last lecture: How to specify the predicate \( s \in L(R) \)
- But yes/no answer is not enough!
- We really want to partition input into tokens
- We adapt regular expressions for this goal
Regular Expressions to Lexical Specifications (1)

- **Step 1:** Write a regular expression for the lexemes of each token
Step 1: Write a regular expression for the lexemes of each token
   - Integer constant: digit\(^+\)
Regular Expressions to Lexical Specifications (1)

- Step 1: Write a regular expression for the lexemes of each token
  - Integer constant: digit^+
  - Identifier: letter (letter + digit)^*
Step 1: Write a regular expression for the lexemes of each token

- Integer constant: digit^+
- Identifier: letter (letter + digit)^*
- Lambda: 'lambda'
Regular Expressions to Lexical Specifications (1)

- **Step 1:** Write a regular expression for the lexemes of each token
  - Integer constant: \( \text{digit}^+ \)
  - Identifier: \( \text{letter} \ (\text{letter} \ + \ \text{digit})^* \)
  - Lambda: 'lambda'
  - ...

Thomas Dillig, CS345H: Programming Languages  Lecture 4: Implementation of Lexical Analysis 5/33
Step 2: Construct R, matching lexemes for all tokens
Step 2: Construct $R$, matching lexemes for all tokens

$R = \text{Integer constant} + \text{Identifier} + \text{Lambda} + \ldots$
Let the input be characters $x_1...x_n$
Let the input be characters $x_1...x_n$

Step 3: For each $1 \leq i \leq n$ check $x_1...x_j \in L(R)$ for some $j$
Let the input be characters $x_1...x_n$

**Step 3:** For each $1 \leq i \leq n$ check $x_1...x_j \in L(R)$ for some $j$

Then, remove $x_1...x_j$ from input and repeat
There are ambiguities in this algorithm. Where?
Ambiguities I

- There are ambiguities in this algorithm. Where?

- How much input is used? What if \( x_1 \ldots x_i \in L(R) \) and \( x_1 \ldots x_j \in L(R) \)?
There are ambiguities in this algorithm. Where?

How much input is used? What if \(x_1 \ldots x_i \in L(R)\) and \(x_1 \ldots x_j \in L(R)\)?

Example: identifier = letter (letter + digit)*, if = 'i' 'f'
Ambiguities I

- There are ambiguities in this algorithm. Where?

- How much input is used? What if \( x_1 \ldots x_i \in L(R) \) and \( x_1 \ldots x_j \in L(R) \)?

- Example: identifier = letter (letter + digit)*, if = 'i' 'f'

- Rule: Pick longest possible string in \( L(R) \)
Ambiguities I

- There are ambiguities in this algorithm. Where?

- How much input is used? What if \( x_1 \ldots x_i \in L(R) \) and \( x_1 \ldots x_j \in L(R) \)?

  - **Example:** identifier = letter (letter + digit)*, if = 'i' 'f'

  - **Rule:** Pick longest possible string in \( L(R) \)

  - This is known as “maximal munch”
What if two rules match with the same number of characters?
Ambiguities II

- What if two rules match with the same number of characters?

- $x_1...x_i \in L(R_1)$ and $x_1...x_i \in L(R_2)$?
Ambiguities II

- What if two rules match with the same number of characters?

- $x_1...x_i \in L(R_1)$ and $x_1...x_i \in L(R_2)$?

- Example: "if"
What if two rules match with the same number of characters?

$x_1...x_i \in L(R_1)$ and $x_1...x_i \in L(R_2)$?

Example: "if"

Rule: Use rule listed first
What if two rules match with the same number of characters?

\[ x_1...x_i \in L(R_1) \text{ and } x_1...x_i \in L(R_2) ? \]

Example: "if"

Rule: Use rule listed first

This is how "if" is matched as a keyword, not identifier
Error Handling

- What if no rule matches a prefix of the input?
Error Handling

- What if no rule matches a prefix of the input?
- **Solution 1:** Get stuck
Error Handling

- What if no rule matches a prefix of the input?
- **Solution 1**: Get stuck $\Rightarrow$ Unacceptable
Error Handling

▶ What if no rule matches a prefix of the input?

▶ **Solution 1**: Get stuck ⇒ Unacceptable

▶ **Better Solution**: Write a rule matching all “bad” strings
Error Handling

- What if no rule matches a prefix of the input?
  - **Solution 1:** Get stuck ⇒ Unacceptable
  
  - **Better Solution:** Write a rule matching all “bad” strings

- **Question:** What kind of rule and where to place it?
Where are we?

- We now know how we can partition input string into tokens assuming we can decide if a string is in the language described by a regular expression.
Where are we?

- We now know how we can partition input string into tokens assuming we can decide if a string is in the language described by a regular expression.

- Next: How to decide if $s \in L(R)$
Finite Automata

- Regular Expressions \(\Leftrightarrow\) Specification
Finite Automata

- Regular Expressions ⇔ Specification
- Finite Automata ⇔ Implementation
Finite Automata

- Regular Expressions ⇔ Specification
- Finite Automata ⇔ Implementation
- A finite automata formally consists of:
Finite Automata

- Regular Expressions $\Leftrightarrow$ Specification
- Finite Automata $\Leftrightarrow$ Implementation
- A finite automata formally consists of:
  - An input alphabet $\Sigma$
Finite Automata

- Regular Expressions $\Leftrightarrow$ Specification
- Finite Automata $\Leftrightarrow$ Implementation
- A finite automata formally consists of:
  - An input alphabet $\Sigma$
  - A set of states $S$

Finite Automata

- Regular Expressions $\Leftrightarrow$ Specification
- Finite Automata $\Leftrightarrow$ Implementation
- A finite automata formally consists of:
  - An input alphabet $\Sigma$
  - A set of states $S$
  - A start state $n$
Finite Automata

- Regular Expressions $\Leftrightarrow$ Specification
- Finite Automata $\Leftrightarrow$ Implementation
- A finite automata formally consists of:
  - An input alphabet $\Sigma$
  - A set of states $S$
  - A start state $n$
  - A set of accepting states $F \subseteq S$
Finite Automata

- Regular Expressions $\Leftrightarrow$ Specification
- Finite Automata $\Leftrightarrow$ Implementation

- A finite automata formally consists of:
  - An input alphabet $\Sigma$
  - A set of states $S$
  - A start state $n$
  - A set of accepting states $F \subseteq S$
  - A set of transitions state $\rightarrow^{\text{input}}$ state
Finite Automata

- Transition $S_1 \xrightarrow{\alpha} S_2$

This means: In state $S_1$ and input character $\alpha$, go to state $S_2$. If end of input and in accepting state $\Rightarrow$ accept. Otherwise $\Rightarrow$ reject.
Finite Automata

- Transition $S_1 \rightarrow^\alpha S_2$

- This means: In state $S_1$ and input character $\alpha$, go to state $S_2$
Finite Automata

- Transition $S_1 \rightarrow^\alpha S_2$

- This means: In state $S_1$ and input character $\alpha$, go to state $S_2$

- If end of input and in accepting state $\Rightarrow$ accept
Finite Automata

- Transition $S_1 \xrightarrow{\alpha} S_2$
- This means: In state $S_1$ and input character $\alpha$, go to state $S_2$
- If end of input and in accepting state $\Rightarrow$ accept
- Otherwise $\Rightarrow$ reject
Finite Automata as State Graphs

- It is much easier to imagine finite automata visually:
It is much easier to imagine finite automata visually:

- A state:
Finite Automata as State Graphs

- It is much easier to imagine finite automata visually:

A state:  

The start state:
Finite Automata as State Graphs

- It is much easier to imagine finite automata visually:

A state:

The start state:

An accepting state:
Finite Automata as State Graphs

- It is much easier to imagine finite automata visually:

A state:

The start state:

An accepting state:

A transition:
A Simple Example

Here is an automaton that only accepts the string "1":

▶
A Simple Example

- Here is an automaton that only accepts the string "1":

![Automaton Diagram](image-url)
Another Simple Example

- A finite automaton accepting any number of 1’s followed by a single 0
Another Simple Example

- A finite automaton accepting any number of 1’s followed by a single 0
- Alphabet: \{0, 1\}
Another Simple Example

- A finite automaton accepting any number of 1’s followed by a single 0

- Alphabet: \{0, 1\}
And Another Example

- Alphabet: \( \{0, 1\} \)
And Another Example

- **Alphabet:** \{0, 1\}

- **What language does this automata recognize?**
And Another Example

- Alphabet: \{0, 1\}

- What language does this automata recognize?
Epsilon Moves

- Another kind of transition: $\varepsilon$-moves
Epsilon Moves

- Another kind of transition: $\varepsilon$-moves

\[ \text{A} \xrightarrow{\varepsilon} \text{B} \]
Epsilon Moves

- Another kind of transition: $\varepsilon$-moves

- Machine can move from state $A$ to $B$ without reading any input
Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)

- Nondeterministic Finite Automata (NFA)
Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - At most one transition per input on any state

- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - At most one transition per input on any state
  - No $\varepsilon$ moves

- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves
Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - At most one transition per input on any state
  - No $\varepsilon$ moves

- Nondeterministic Finite Automate (NFA)
Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - At most one transition per input on any state
  - No $\varepsilon$ moves

- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
Deterministic and Nondeterministic Automata

- **Deterministic Finite Automata (DFA)**
  - At most one transition per input on any state
  - No $\varepsilon$ moves

- **Nondeterministic Finite Automate (NFA)**
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves
Execution of Finite Automata

- A DFA can only take one path through the state graph that is completely determined by the input.
Execution of Finite Automata

- A DFA can only take one path through the state graph that is completely determined by the input

- NFAs can choose:
  - Whether to make $\varepsilon$ moves
  - Which one of multiple transitions for a single input to take
Execution of Finite Automata

- A DFA can only take one path through the state graph that is completely determined by the input.

- NFAs can choose:
  - Whether to make $\varepsilon$ moves.
Execution of Finite Automata

- A DFA can only take one path through the state graph that is completely determined by the input.

- NFAs can choose:
  - Whether to make \( \varepsilon \) moves
  - Which one of multiple transitions for a single input to take
Acceptance of NFAs

► This means: A NFA can get into multiple states at the same time
Acceptance of NFAs

- This means: A NFA can get into multiple states at the same time

- Consider again the alphabet $\Sigma = \{0, 1\}$ and the language of all strings ending in at least two 0s.
Acceptance of NFAs

- This means: A NFA can get into multiple states at the same time

- Consider again the alphabet $\Sigma = \{0, 1\}$ and the language of all strings ending in at least two 0s.
Acceptance of NFAs

- This means: A NFA can get into multiple states at the same time

- Consider again the alphabet $\Sigma = \{0, 1\}$ and the language of all strings ending in at least two 0s.

- Consider input 1 0 0

![NFA Diagram]

Rule: NFA accepts if it can get to a final state.
Acceptance of NFAs

- This means: A NFA can get into multiple states at the same time

- Consider again the alphabet $\Sigma = \{0, 1\}$ and the language of all strings ending in at least two 0s.

- Consider input 1 0 0
Acceptance of NFAs

- This means: A NFA can get into multiple states at the same time

- Consider again the alphabet $\Sigma = \{0, 1\}$ and the language of all strings ending in at least two 0s.

- Consider input 1 0 0

![Diagram of an NFA with states and transitions]
Acceptance of NFAs

- This means: A NFA can get into multiple states at the same time

- Consider again the alphabet $\Sigma = \{0, 1\}$ and the language of all strings ending in at least two 0s.

- Consider input 1 0 0

![NFA Diagram]
Acceptance of NFAs

- This means: A NFA can get into multiple states at the same time

- Consider again the alphabet $\Sigma = \{0, 1\}$ and the language of all strings ending in at least two 0s.

- Consider input 1 0 0

- Rule: NFA accepts if it can get to a final state
NFAs vs. DFAs

- **Fundamental Result**: NFAs and DFAs recognize the same set of languages (regular languages)
NFAs vs. DFAs

- **Fundamental Result:** NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are faster to execute, since there are no choices to consider
NFAs vs. DFAs

- **Fundamental Result:** NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are faster to execute, since there are no choices to consider

- But NFAs can be much simpler for the same language
NFAs vs. DFAs

- **Fundamental Result:** NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are faster to execute, since there are no choices to consider

- But NFAs can be much simpler for the same language

- **Result:** DFAs can be exponentially larger than NFA recognizing same language
Regular Expressions to Finite Automata

- High-Level Sketch:
Regular Expressions to Finite Automata

- High-Level Sketch:
  - Lexical Specification
Regular Expressions to Finite Automata

▶ High-Level Sketch:
  ▶ Lexical Specification
  ▶ Regular Expressions
Regular Expressions to Finite Automata

- High-Level Sketch:
  - Lexical Specification
  - Regular Expressions
  - NFA
Regular Expressions to Finite Automata

- High-Level Sketch:
  - Lexical Specification
  - Regular Expressions
  - NFA
  - DFA

⇒ Implementation of DFA

Lexer
Regular Expressions to Finite Automata

- High-Level Sketch:
  - Lexical Specification
  - Regular Expressions
  - NFA
  - DFA
  - Implementation of DFA
Regular Expressions to Finite Automata

- **High-Level Sketch:**
  - Lexical Specification
  - Regular Expressions
  - NFA
  - DFA
  - Implementation of DFA

⇒ Lexer
For each kind of regular expression, define an NFA and combine
Regular Expressions to NFA (1)

- For each kind of regular expression, define an NFA and combine

- Will use the following notation: NFA for regular expression $M$:
Regular Expressions to NFA (1)

- For each kind of regular expression, define an NFA and combine

- Will use the following notation: NFA for regular expression $M$:

- Base cases:
Regular Expressions to NFA (1)

- For each kind of regular expression, define an NFA and combine

- Will use the following notation: NFA for regular expression $M$:

- Base cases:
  - For $\varepsilon$:
Regular Expressions to NFA (1)

- For each kind of regular expression, define an NFA and combine

- Will use the following notation: NFA for regular expression $M$:

- Base cases:
  - For $\varepsilon$:
  - For input $a$:
Regular Expressions to NFA (2)

- For $AB$:

![Diagram of NFA for AB]
Regular Expressions to NFA (2)

- For $AB$:

- For $A + B$:
Regular Expressions to NFA (3)

For $A^*$:
Consider the regular expression \((1 + 0)^*1\)
Example of Regular Expression to NFA conversion

- Consider the regular expression \((1 + 0)^*1\)
Example of Regular Expression to NFA conversion

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

Diagram:

- States: C, D, E, F
- Transitions:
  - C to E on 1
  - D to F on 0

Example of Regular Expression to NFA conversion

- Consider the regular expression \((1 + 0)^*1\)
Example of Regular Expression to NFA conversion

Consider the regular expression \((1 + 0)^*1\)
Example of Regular Expression to NFA conversion

Consider the regular expression \((1 + 0)^*1\)
NFA to DFA: The Trick

- **Insight:** Simulate the NFA
NFA to DFA: The Trick

- **Insight:** Simulate the NFA

- At any given time, the NFA is in a set of states
NFA to DFA: The Trick

- **Insight**: Simulate the NFA

- At any given time, the NFA is in a set of states

- States in the DFA \(\Rightarrow\) all (reachable) subsets of states in the NFA
NFA to DFA: The Trick

- **Insight:** Simulate the NFA

- At any given time, the NFA is in a *set of states*

- States in the DFA $\Rightarrow$ all (reachable) subsets of states in the NFA

- **Start State:**
NFA to DFA: The Trick

- **Insight:** Simulate the NFA

- At any given time, the NFA is in a set of states

- States in the DFA ⇒ all (reachable) subsets of states in the NFA

- **Start State:** the set of states reachable through $\varepsilon$ moves from the NFA start state
NFA to DFA: The Trick

- **Insight:** Simulate the NFA

- At any given time, the NFA is in a set of states

- States in the DFA ⇒ all (reachable) subsets of states in the NFA

- **Start State:** the set of states reachable through \( \varepsilon \) moves from the NFA start state

- Add transition \( A \rightarrow^a B \) to DFA iff:
NFA to DFA: The Trick

- **Insight:** Simulate the NFA

- At any given time, the NFA is in a set of states

- States in the DFA $\Rightarrow$ all (reachable) subsets of states in the NFA

- **Start State:** the set of states reachable through $\varepsilon$ moves from the NFA start state

- Add transition $A \xrightarrow{a} B$ to DFA iff:
  - $B$ is in the set of states reachable from any state in $A$ after seeing input $a$, considering $\varepsilon$ moves as well
NFA to DFA: Example

Recall our friendly NFA for \((1 + 0)^*1:\)
Recall our friendly NFA for \((1 + 0)^*1:\)
Recall our friendly NFA for \((1 + 0)^*1:\)
Recall our friendly NFA for \((1 + 0)^*1:\)
Recall our friendly NFA for \((1 + 0)^*1:\)
NFA to DFA: Example

Recall our friendly NFA for \((1 + 0)^*1:\)

\[
\begin{array}{cccccccccc}
A & B & C & D & E & F & G & H & I & J \\
\epsilon & \epsilon & 1 & \epsilon & \epsilon & \epsilon & \epsilon & \epsilon & 1 & \epsilon \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
A & B & C & D & E & F & G & H & I & J \\
\epsilon & \epsilon & \epsilon & 0 & \epsilon & \epsilon & \epsilon & \epsilon & \epsilon & \epsilon \\
\end{array}
\]
Recall our friendly NFA for \((1 + 0)^*1:\)
NFA to DFA: Example

Recall our friendly NFA for \((1 + 0)^*1:\)

![NFA Diagram]

- States: A, B, C, D, E, F, G, H, I, J
- Transitions:
  - \(\varepsilon\) transitions:
    - From A to B, C, E, D, F, G, H, I, J
  - 1 transition:
    - From F to I
  - 0 transition:
    - From D to G, B to C
- Initial state: A
- Accepting state: J
NFA to DFA: Example

Recall our friendly NFA for \((1 + 0)^*1\):
NFA to DFA: Example

Recall our friendly NFA for \((1 + 0)^*1:\)

![NFA Diagram]

- States: A, B, C, D, E, F, G, H, I, J
- Transitions:
  - \(A \xrightarrow{\varepsilon} B, B \xrightarrow{\varepsilon} A, B \xrightarrow{1} C, C \xrightarrow{1} E, E \xrightarrow{\varepsilon} C, C \xrightarrow{\varepsilon} D, D \xrightarrow{0} F, F \xrightarrow{\varepsilon} C, E \xrightarrow{\varepsilon} G, G \xrightarrow{\varepsilon} H, H \xrightarrow{\varepsilon} I, I \xrightarrow{1} J\)

Accepting states: J

ABCDFGHI

ABCDHI

0

1
NFA to DFA: Example

Recall our friendly NFA for \((1 + 0)^*1\):

\[
\begin{array}{c}
\text{A} \\
\text{B} \\
\text{C} \\
\text{D} \\
\text{E} \\
\text{F} \\
\text{G} \\
\text{H} \\
\text{I} \\
\text{J}
\end{array}
\]

\[
\begin{array}{c}
\text{ABCDFGHI} \\
\text{ABCDFGHI}
\end{array}
\]
NFA to DFA: Example

Recall our friendly NFA for $(1 + 0)^* 1$:
Recall our friendly NFA for \((1 + 0)^*1:\)
NFA to DFA: Example

Recall our friendly NFA for \((1 + 0)^*1\):
Recall our friendly NFA for \((1 + 0)^*1\):

\[
\begin{array}{c}
A \xrightarrow{\varepsilon} B \xrightarrow{\varepsilon} C \xrightarrow{1} E \xrightarrow{\varepsilon} G \xrightarrow{\varepsilon} H \xrightarrow{\varepsilon} I \xrightarrow{1} J \\
D \xrightarrow{\varepsilon} F \xrightarrow{\varepsilon} G \xrightarrow{\varepsilon} H \xrightarrow{\varepsilon} I \xrightarrow{1} J
\end{array}
\]
Recall our friendly NFA for \((1 + 0)^*1:\)
NFA to DFA: Example

Recall our friendly NFA for $(1 + 0)^*1$:
Recall our friendly NFA for \((1 + 0)^*1:\)

![Diagram of NFA and DFA conversion]
Recall our friendly NFA for $(1 + 0)^*1$:
Recall our friendly NFA for $(1 + 0)^*1$: 
Recall our friendly NFA for \((1 + 0)^*1:\)
Recall our friendly NFA for \((1 + 0)^*1:\)
NFA to DFA: Example

Recall our friendly NFA for $(1 + 0)^*1$:
NFA to DFA: Example

Recall our friendly NFA for \((1 + 0)^*1:\)

![Diagram of an NFA with states A, B, C, D, E, F, G, H, I, and J, and transitions for the characters 0 and 1.](image)

States:
- A
- B
- C
- D
- E
- F
- G
- H
- I
- J

Transitions:
- \(\epsilon\) from A to B
- 1 from B to C
- 0 from D to F
- \(\epsilon\) from E to G
- \(\epsilon\) from G to H
- 1 from I to J
- 0 from ABCDFGHI to ABCDFGHI
- 1 from ABCDEGHIJ to ABCDEGHIJ

Accepting States:
- J
NFA to DFA: Example

Recall our friendly NFA for \((1 + 0)^*1:\)

\[ \begin{array}{c}
\varepsilon \\
B & \varepsilon \\
C & 1 \\
D & \varepsilon \\
E & \varepsilon \\
G & \varepsilon \\
H & \varepsilon \\
I & 1 \\
J \\
\end{array} \]
Recall our friendly NFA for \((1 + 0)^*1\):
NFA to DFA: How many states?

- We need a state in the DFA for each set of states the NFA can be in.
NFA to DFA: How many states?

- We need a state in the DFA for each set of states the NFA can be in
- How many different states?
NFA to DFA: How many states?

- We need a state in the DFA for each set of states the NFA can be in.

- How many different states?

- If there are $N$ states, the NFA must be in some subset of those $N$ states.
NFA to DFA: How many states?

- We need a state in the DFA for each set of states the NFA can be in.
- How many different states?
- If there are $N$ states, the NFA must be in some subset of those $N$ states.
- How many subsets of $N$ states?
NFA to DFA: How many states?

- We need a state in the DFA for each set of states the NFA can be in.
- How many different states?
- If there are $N$ states, the NFA must be in some subset of those $N$ states.
- How many subsets of $N$ states? $2^N$
Implementation

- A DFA can be implemented by a 2D table $T$.
A DFA can be implemented by a 2D table $T$:
- One dimension is “states”
Implementation

- A DFA can be implemented by a 2D table $T$
  - One dimension is “states”
  - Other dimension is “input symbols”
Implementation

- A DFA can be implemented by a 2D table $T$
  - One dimension is “states”
  - Other dimension is “input symbols”
  - For every transition $A \rightarrow^c B$, define $T[A, c] = B$
Implementation

- A DFA can be implemented by a 2D table $T$
  - One dimension is “states”
  - Other dimension is “input symbols”
  - For every transition $A \xrightarrow{c} B$, define $T[A, c] = B$

- DFA “execution”: If in state $A$ and input $c$, read $T[A, c] = B$ and skip to state $B$
Implementation

- A DFA can be implemented by a 2D table $T$
  - One dimension is “states”
  - Other dimension is “input symbols”
  - For every transition $A \rightarrow^c B$, define $T[A, c] = B$
- DFA “execution”: If in state $A$ and input $c$, read $T[A, c] = B$ and skip to state $B$
- Very efficient
Table Implementation of a DFA
Table Implementation of a 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>
Writing regular expressions as NFAs and converting them to DFAs is exactly what flex does.
Writing regular expressions as NFAs and converting them to DFAs is exactly what flex does.

In fact, if you open the auto-generated flex file lex.yy.c, you will see these tables emitted.
Writing regular expressions as NFAs and converting them to DFAs is exactly what `flex` does.

In fact, if you open the auto-generated `flex` file `lex.yy.c`, you will see these tables emitted.

But, these DFAs can be huge.
Implementation cont.

- Writing regular expressions as NFAs and converting them to DFAs is exactly what `flex` does.

- In fact, if you open the auto-generated `flex` file `lex.yy.c`, you will see these tables emitted.

- But, these DFAs can be huge.

- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations.