

# Implementation of Lexical Analysis

CPSC 310 - Programming Languages

# Outline

- Specify lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata (DFA)
  - Non-deterministic Finite Automata (NFA)
- Implementation of regular expressions
  - Regular expression  $\rightarrow$  NFA  $\rightarrow$  DFA  $\rightarrow$  Tables

# Regular Expressions in Lexical Specification

- A regular expression specifies a predicate  $s \in L(R)$ , that is, is a string  $s$  a member of the language  $L(R)$
- Testing for set membership is not enough, we need to partition the input into tokens
- We can adapt regular expressions to meet this goal

# Regular Expressions to Lexical Specification

- 1 Select a set of tokens
  - Integer, Keyword, Identifier, ...
- 2 Write a regular expression (or rule) for the lexemes of each token
  - Integer =  $[0123456789]^+$
  - Keyword =  $(if \mid else \mid \dots)$
  - Identifiers:  $[A - Za - z] ([A - Za - z] \mid [0123456789])^*$

# Regular Expressions to Lexical Specification

- 3 Construct a regular expression that matches all lexemes for all tokens
  - $R = \text{Integer} \mid \text{Keyword} \mid \text{Identifier} \mid \dots$
  - $R = R_1 \mid R_2 \mid R_3 \mid \dots$
- If  $s \in L(R)$  then  $s$  is a lexeme
  - Furthermore  $s \in L(R_i)$  for some  $i$
  - This  $i$  determines the token that is reported

# Regular Expressions to Lexical Specification

- 4 Let the input be  $x_1 \dots x_n$ 
  - $x_1 \dots x_n$  are characters
  - For  $1 \leq i \leq n$  check if  $x_1 \dots x_i \in L(R)$
  - If so, it must be that  $x_1 \dots x_i \in L(R_j)$  for some  $j$
  - Otherwise,  $s \notin L(R)$
- 5 Remove  $x_1 \dots x_i$  from the input and got to the previous step

# Options for Handling Whitespace and Comments

## 1 We could create a token for whitespace or comments

- Whitespace = (' ' | '\n' | '\t')+
- Comment = ...
- An input of " \t\n 42 " is transformed to the token stream  
Whitespace Integer Whitespace

## 2 The lexer skips whitespace and comments

- This is the preferred method because whitespace and comments are irrelevant to the parser (for most languages)
- The lexer still needs to match a whitespace (or comment) regular expression, but a token is not output

# Ambiguities

- How much input is used?
  - What if  $x_1 \dots x_i \in L(R)$  and  $x_1 \dots x_k \in L(R)$ ?
  - Rule: choose the longest possible substring (“maximal munch”)
- Which token is used?
  - What if  $x_1 \dots x_i \in L(R_j)$  and  $x_1 \dots x_i \in L(R_k)$ ?
  - Rule: choose the rule listed first ( $j$  if  $j < k$ )

# Error Handling

- What if no regular expression matches a prefix of the input?
- Problem: the algorithm needs to terminate
- Solution: write a rule matching all invalid strings and place it at the end of the rules
- Lexer tools allow you to write
$$R = R_1 \mid \dots \mid \text{Error}$$
where the token `Error` matches if nothing else matches

# Summary

- Regular expressions provide a concise notation for string patterns
- Adapting regular expressions to lexical analysis requires small extensions to resolve ambiguities and handle errors
- Good algorithms are known that
  - Require only a single pass over the input
  - Require few operations per character (table lookup)

# Regular Languages and Finite Automata

- Result from formal language theory: regular expressions and finite automata both define the class of regular languages
- Thus, lexical analysis uses:
  - Regular expressions for specification
  - Finite automata for implementation (automatic generation of lexical analyzers)

# Finite Automata

- A finite automata is a *recognizer* for the set of strings of a regular language
- A finite automaton consists of 5-tuple  $(\Sigma, Q, q_0, F, \delta)$ :
  - $\Sigma$ : a finite input alphabet
  - $Q$ : a set of states
  - $q_0$ : a start state
  - $F \subseteq S$ : a set of accepting states
  - $\delta : Q \times \Sigma \rightarrow Q$ : a set of transitions

# Finite Automata

- Transition notation

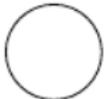
$$s_1 \xrightarrow{a} s_2$$

is read: in state  $s_1$  on input  $a$  go to state  $s_2$

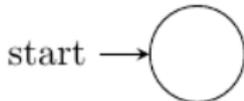
- Each transition “consumes” a character from the input
- At the end of input (or no transition possible)
  - If in accepting state, accept ( $s \in L(R)$ )
  - Otherwise, reject ( $s \notin L(R)$ )

# Finite Automata State Graphs

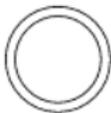
- A state:



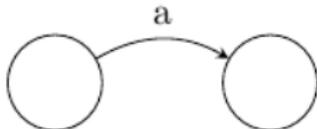
- A start state:



- An accepting state:



- A transition:



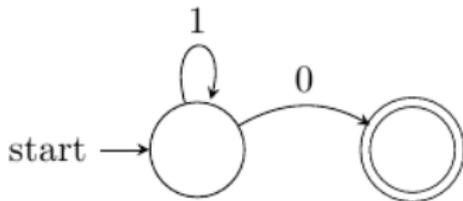
# A Simple Example

- A finite automaton that accepts only "1"



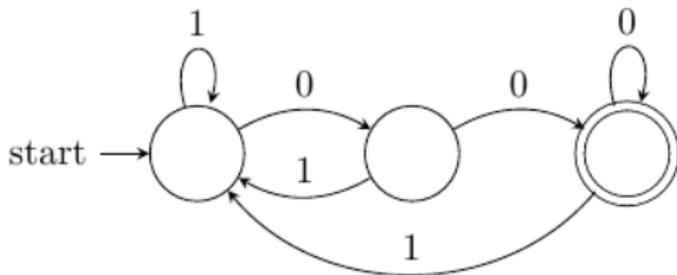
# Another Simple Example

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



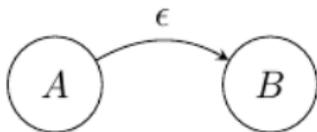
# Another Example

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



# Epsilon Transitions

- Epsilon transitions:
- The automaton can move from state  $A$  to state  $B$  without consuming input



# Deterministic and Non-Deterministic Automata

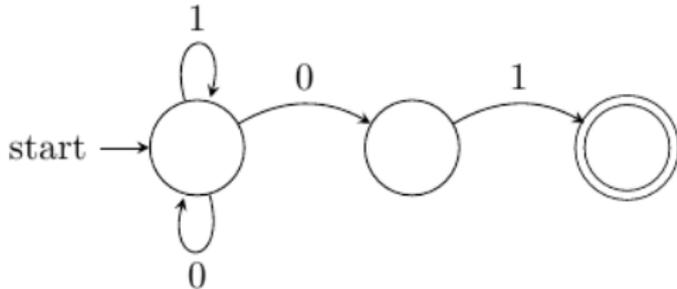
- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No epsilon transitions
- Non-deterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have epsilon transitions
- Finite automata have finite memory – only enough to encode the current state

# Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose:
  - whether to make epsilon transitions
  - which of multiple transitions for a single input to take

# Acceptance of NFAs

- An NFA can get into multiple states
- An NFA accepts an input if it can get in a final state
- Exampe input: 1 0 1



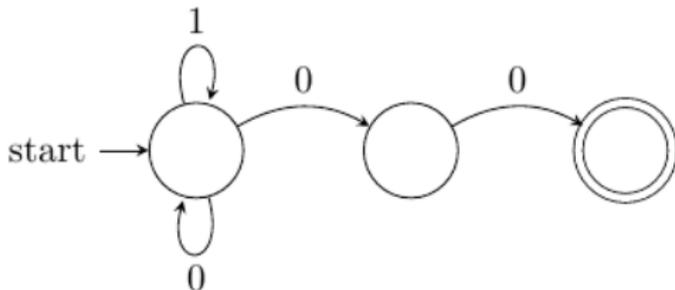
# NFA versus DFA

- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are easier to implement
- A DFA can be exponentially larger than an equivalent NFA

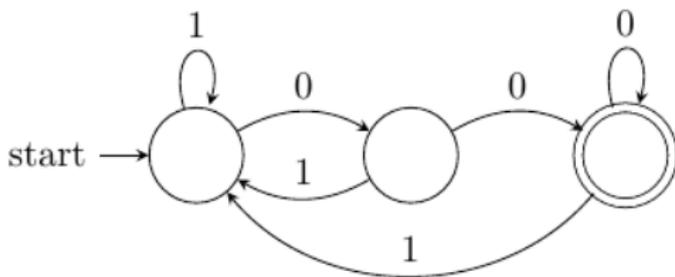
# NFA versus DFA

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

- NFA:



- DFA:

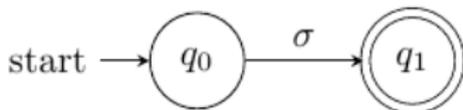


# Regular Expressions to Finite Automata

- The implementation of a lexical specification as a finite automata has the following transformations:
  - 1 Lexical specification
  - 2 Regular expressions
  - 3 NFA
  - 4 DFA
  - 5 Table driven implementation of DFA

# Regular Expressions to NFA

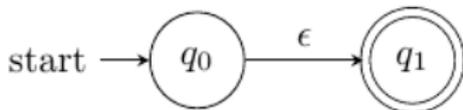
- We can define an NFA for each basic regular expression and then connect the NFAs together based on the operators
- Base case  $\sigma$ :



- $\langle \sigma \rangle = (\{\sigma\}, \{q_0, q_1\}, q_0, \{q_1\}, \{(q_0, \sigma, q_1)\})$

# Regular Expressions to NFA

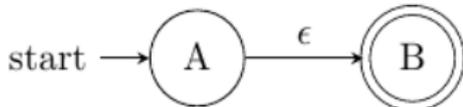
- Base case  $\epsilon$ :



- $\langle \epsilon \rangle = (\emptyset, \{q_0, q_1\}, q_0, \{q_1\}, \{(q_0, \epsilon, q_1)\})$

# Regular Expressions to NFA

- $AB$ : make an  $\epsilon$  transition from the accepting state of  $A$  to start state of  $B$



- Inductive case concatenation:

$$\langle A \rangle = (\Sigma_A, Q_A, q_A, \{f_A\}, \sigma_A)$$

$$\langle B \rangle = (\Sigma_B, Q_B, q_B, \{f_B\}, \sigma_B)$$

$$\langle AB \rangle = (\Sigma_A \cup \Sigma_B,$$

$$Q_A \cup Q_B,$$

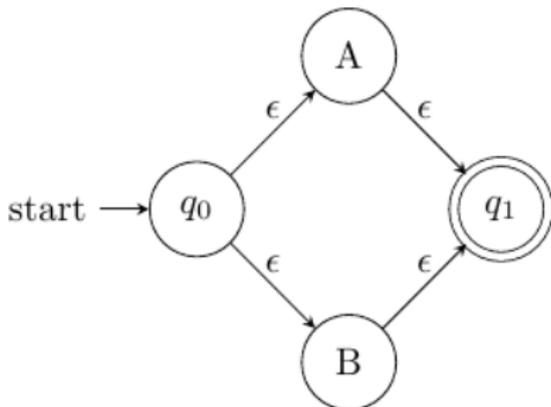
$$q_A,$$

$$\{f_B\},$$

$$\sigma_A \cup \sigma_B \cup \{(f_A, \epsilon, q_B)\})$$

# Regular Expressions to NFA

- $A|B$ : create a new start state and add  $\epsilon$  transitions from the new start state to the start states of  $A$  and  $B$ , then create a new accepting state and add  $\epsilon$  transitions from the accepting states of  $A$  and  $B$  to the new accepting state



# Regular Expressions to NFA

- Inductive case union:

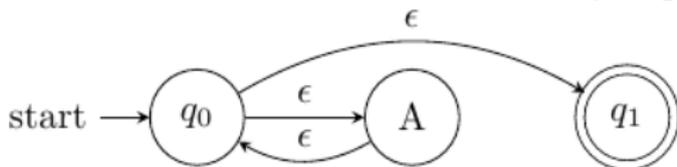
$$\langle A \rangle = (\Sigma_A, Q_A, q_A, \{f_A\}, \sigma_A)$$

$$\langle B \rangle = (\Sigma_B, Q_B, q_B, \{f_B\}, \sigma_B)$$

$$\begin{aligned} \langle A|B \rangle = & (\Sigma_A \cup \Sigma_B, \\ & Q_A \cup Q_B \cup \{q_0, q_1\}, \\ & q_0, \\ & \{q_1\}, \\ & \sigma_A \cup \sigma_B \cup \{(q_0, \epsilon, q_A), (q_0, \epsilon, q_B), (f_A, \epsilon, q_1), (f_B, \epsilon, q_1)\}) \end{aligned}$$

# Regular Expressions to NFA

- $A^*$ : create a new start state and accepting state and add an  $\epsilon$  transitions: from the new start state to the start state of  $A$ , from the accepting state of  $A$  to the new start state, and from the new start state to the new accepting state.



# Regular Expressions to NFA

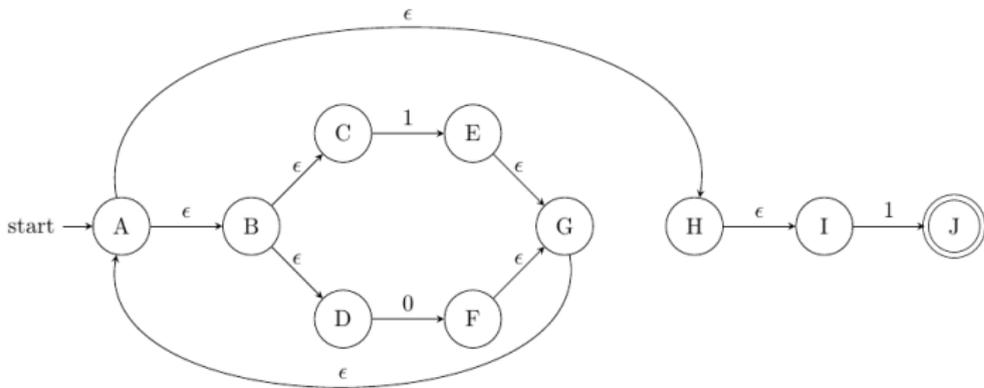
- Inductive case Kleene star:

$$\langle A \rangle = (\Sigma_A, Q_A, q_A, \{f_A\}, \sigma_A)$$

$$\begin{aligned} \langle A^* \rangle = & (\Sigma_A, \\ & Q_A \cup \{q_0, q_1\}, \\ & q_0, \\ & \{q_1\}, \\ & \sigma_A \cup \{(q_0, \epsilon, q_A), (f_A, \epsilon, q_0), (q_0, \epsilon, q_1)\}) \end{aligned}$$

# Regular Expressions to NFA Example

- Consider the regular expression:  $(1|0)^* 1$
- The NFA is



# NFA to DFA (The Trick)

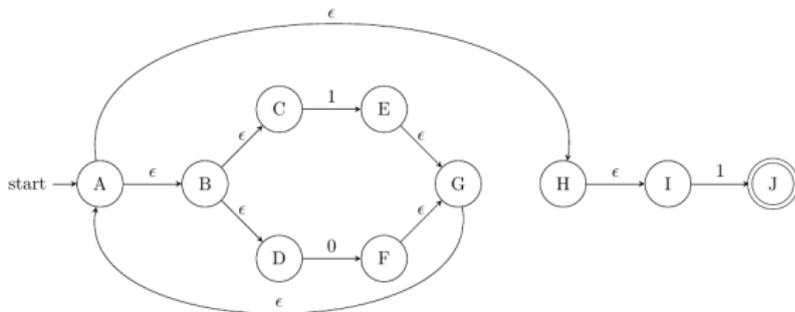
- Simulate the NFA
- Each state of the DFA is a non-empty subset of states of the NFA
- The start state is the set of NFA states reachable through epsilon transitions from the NFA start state
- Add a transition  $S \xrightarrow{a} S'$  to the DFA if and only if  $S'$  is the set of NFA states reachable from any state in  $S$  after seeing the input  $a$  (considering epsilon transitions as well)

# NFA to DFA Remark

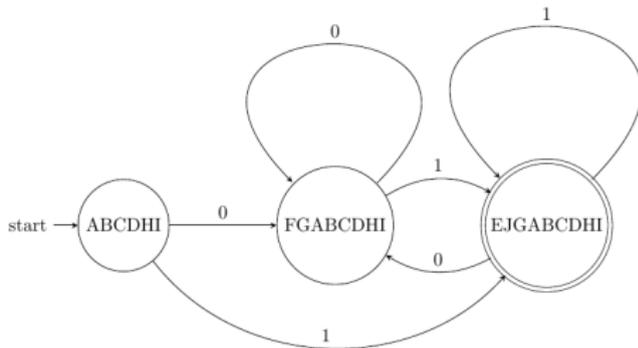
- An NFA may be in many states at any time
- If there are  $N$  states, the NFA must be in some subset of those  $N$  states
- There are  $2^N - 1$  possible subsets (finitely many)

# NFA to DFA Example

## ■ NFA



## ■ DFA



# Subset Construction

- The subset construction algorithm converts an NFA to a DFA
- Algorithm
  - input: NFA =  $(\Sigma, Q, q_0, F, \delta)$
  - output: DFA =  $(\Sigma, Q', q'_0, F', \delta')$
  - subroutines, where  $p$  is an NFA state:
    - $\epsilon$ -closure( $\delta, p$ ) (and  $\epsilon$ -closure( $\delta, Q$ ))
    - transition( $\delta, p, \sigma$ ) (and transition( $\delta, Q, \sigma$ ))

# $\epsilon$ -closure

- $\epsilon$ -closure( $\delta, p$ )
  - set of states  $q$  such that  $p \rightarrow^\epsilon q$
  - $\epsilon$ -closure( $\delta, p$ ) =  $\{q \mid p \rightarrow^\epsilon q \in \delta\}$
  - $\epsilon$ -closure( $\delta, Q$ ) =  $\{q \mid p \in Q, p \rightarrow^\epsilon q \in \delta\}$
- Notes:
  - $\epsilon$ -closure( $\delta, p$ ) always includes  $p$
  - $\epsilon$ -closure( $\delta$ ) or  $\epsilon$ -closure( $Q$ ) is sometimes used as a shorthand when  $\delta$  is clear from the context.

# $\epsilon$ -closure Algorithm Approach

- Input: NFA =  $(\Sigma, Q, q_0, F, \delta)$  and state set  $R$
- Output: state set  $R'$

- Algorithm:

$R' \leftarrow R$

**repeat**

$R \leftarrow R'$

$R' \leftarrow R \cup \{q \mid p \in R, (p, \epsilon, q) \in \delta\}$

**until**  $R = R'$

# transition( $\delta, p, \sigma$ )

- transition( $\delta, p, \sigma$ )
  - Set of states  $q$  such that  $\langle p, \sigma, q \rangle \in \delta$
  - transition( $\delta, p, \sigma$ ) =  $\{q \mid \langle p, \sigma, q \rangle \in \delta\}$
  - transition( $\delta, Q, \sigma$ ) =  $\{q \mid p \in Q, \langle p, \sigma, q \rangle \in \delta\}$ 
    - that is, can “lift” transition to a set of states
- Notes
  - transition( $\delta, p, \sigma$ ) is  $\emptyset$  if no transition  $\langle p, \sigma, q \rangle \in \delta$  for any  $q$
  - transition( $p, \sigma$ ) or transition( $R, \sigma$ ) is sometimes used as a shorthand when  $\delta$  is clear from the context

# NFA to DFA Reduction Algorithm

## ■ Algorithm:

$r_0 \leftarrow \epsilon\text{-closure}(\delta, q_0)$

$R \leftarrow \{r_0\}$

▷ DFA start state

**while**  $\exists$  an unmarked state  $r \in R$  **do**

mark  $r$

▷ each state visited once

**for all**  $\sigma \in \Sigma$  **do**

$E \leftarrow \text{transition}(\delta, r, \sigma)$

▷ states reached via  $\sigma$

$e \leftarrow \epsilon\text{-closure}(\delta, E)$

▷ states reached via  $\epsilon$

**if**  $e \notin R$  **then**

▷ if the state  $e$  is new

$R \leftarrow R \cup \{e\}$

▷ add  $e$  to  $R$  unmarked

$\delta' \leftarrow \delta' \cup \{\langle r, \sigma, e \rangle\}$

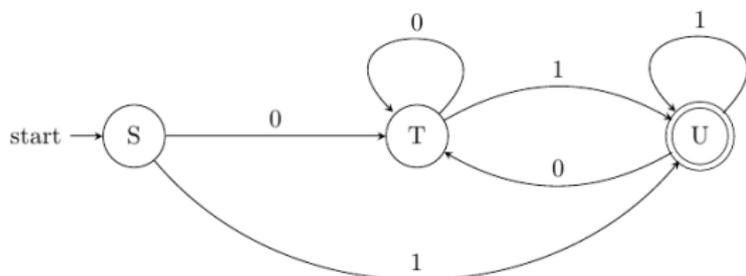
$Q' \leftarrow R$

$F' \leftarrow \{r \mid \exists s \in r \text{ where } s \in F\}$

# DFA Table Implementation

- A DFA can be implemented by a 2D table  $T$ 
  - One dimension is “states”
  - The other dimension is “input symbols”
  - For every transition  $S_i \xrightarrow{a} S_k$  define  $T[i, a] = k$
- DFA “execution”
  - If in state  $S_i$  and input  $a$ , then read  $T[i, a]k$  and skip to state  $S_k$
  - This is efficient

# Example: Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

# Implementation Continued

- The NFA to DFA conversion is the core operation of lexical analysis 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

# Theory versus Practice

- DFAs *recognize* lexemes. A lexer must return a *type of acceptance* (token type) rather than simply an accept/reject indication
- DFAs consume the complete string and accept or reject it. A lexer must *find* the end of the lexeme in the input stream and then find the *next* one, etc.