Compiler Design

Lecture 5: Lexical Analysis IV & Syntax Analysis

Sahar Selim

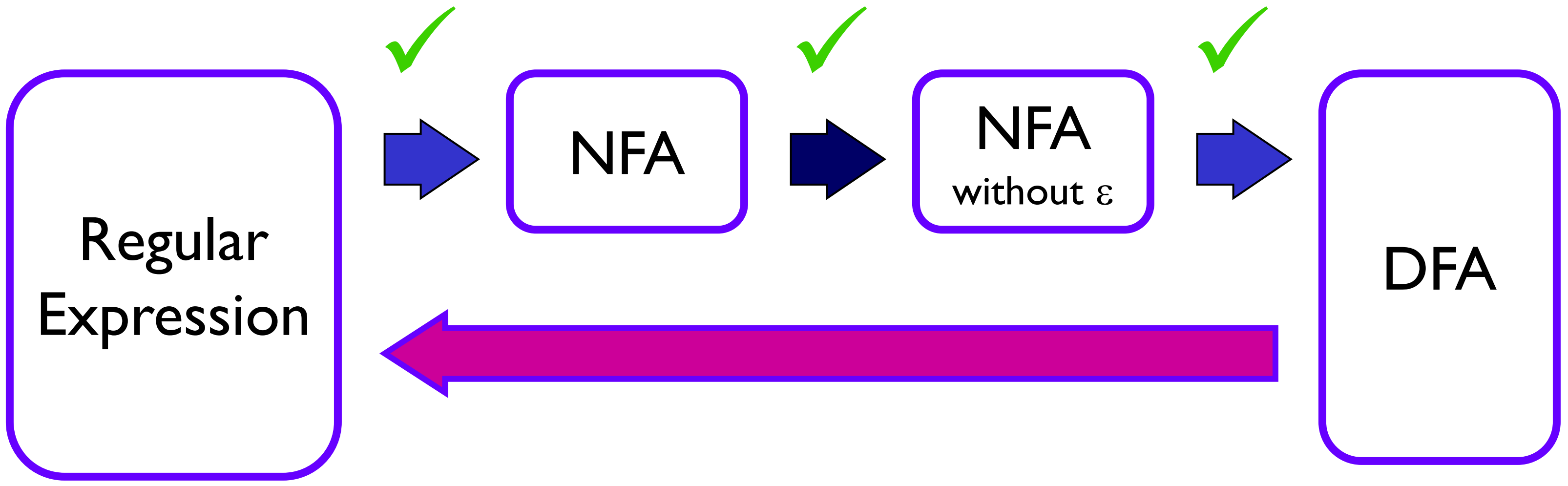
Agenda

1. Lexical Analysis
   - Conversion from DFA to Regular Expression
2. Syntax Analysis
   - The Parsing process
   - Context free grammar (CFG)
     - Comparison to Regular Expression Notation
     - Recursion in CFG

1 Lexical Analysis

From DFA to Regular Expression

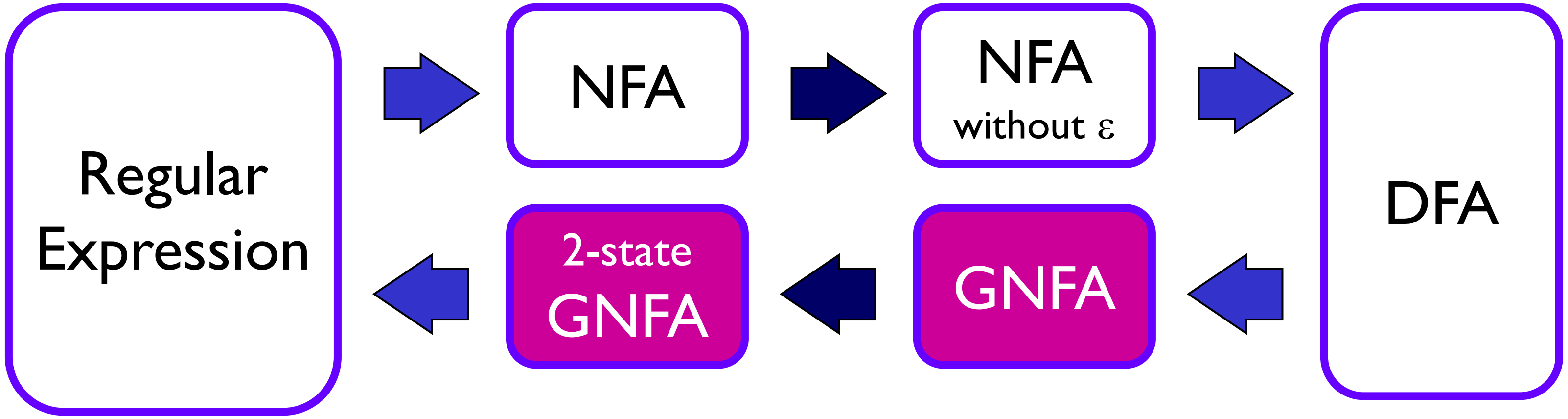
Road map



Converting DFA to Regular Expression

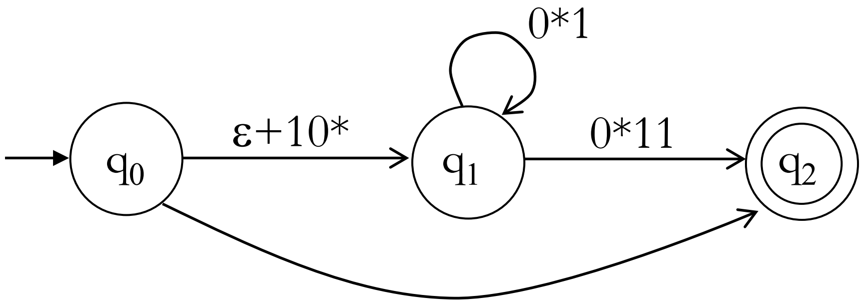
- Arden's Method
- State Elimination Method

Conversion from DFA to RE



Generalized NFAs

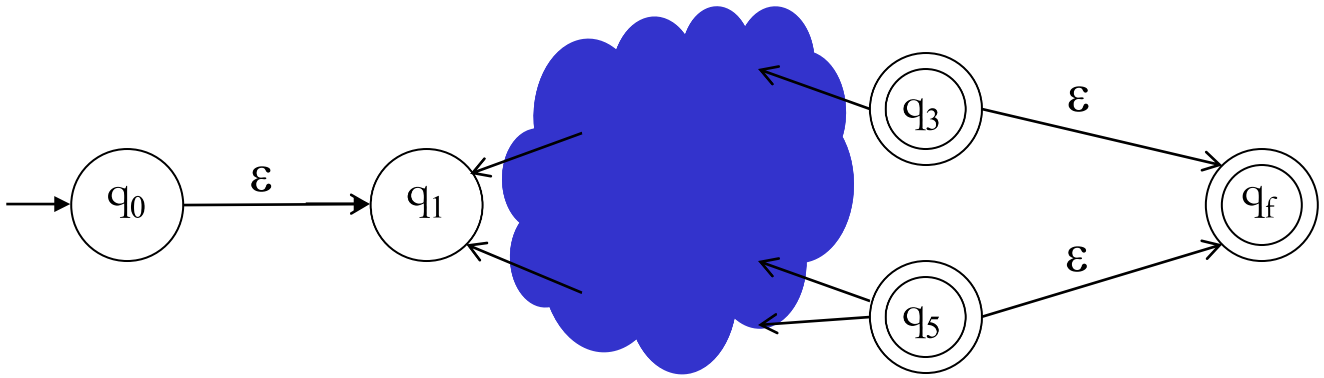
- A generalized NFA is an NFA whose transitions are labeled by regular expressions, like



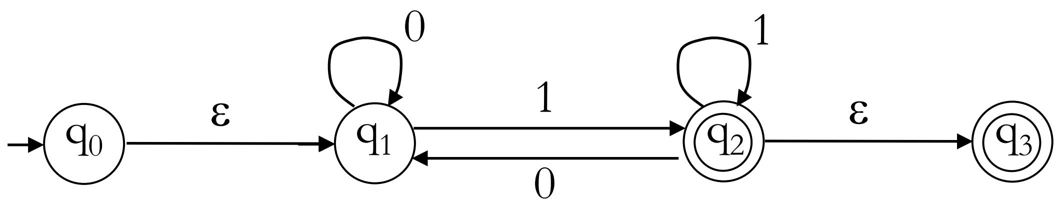
moreover

- It has exactly one accept state, different from its start state
- No arrows come into the start state
- No arrows go out of the accept state

Converting a DFA to a GNFA

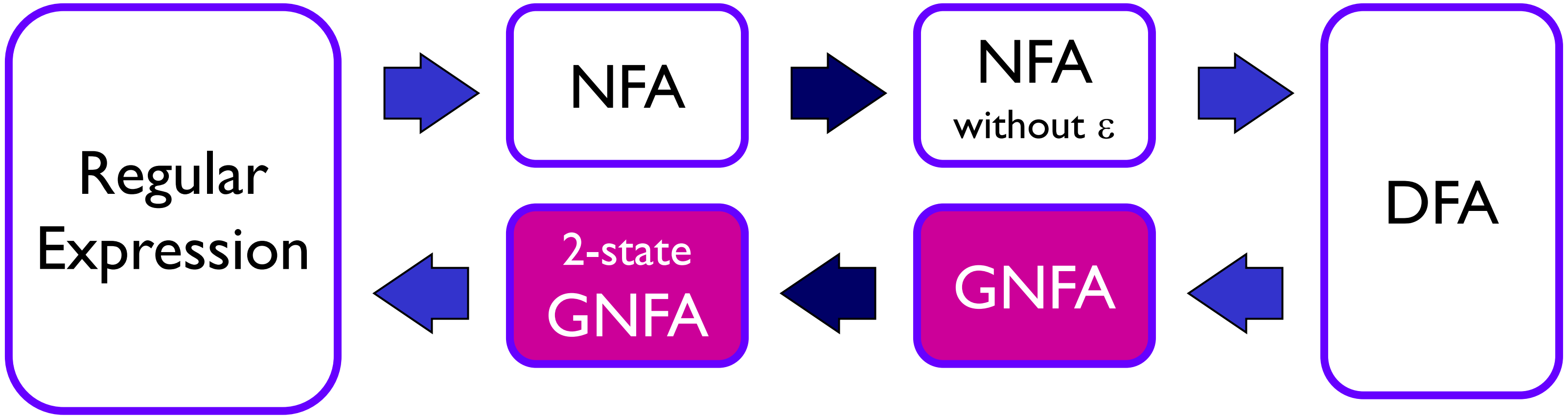


Conversion example



- ✓ It has exactly one accept state, different from its start state
- ✓ No arrows come into the start state
- ✓ No arrows go out of the accept state

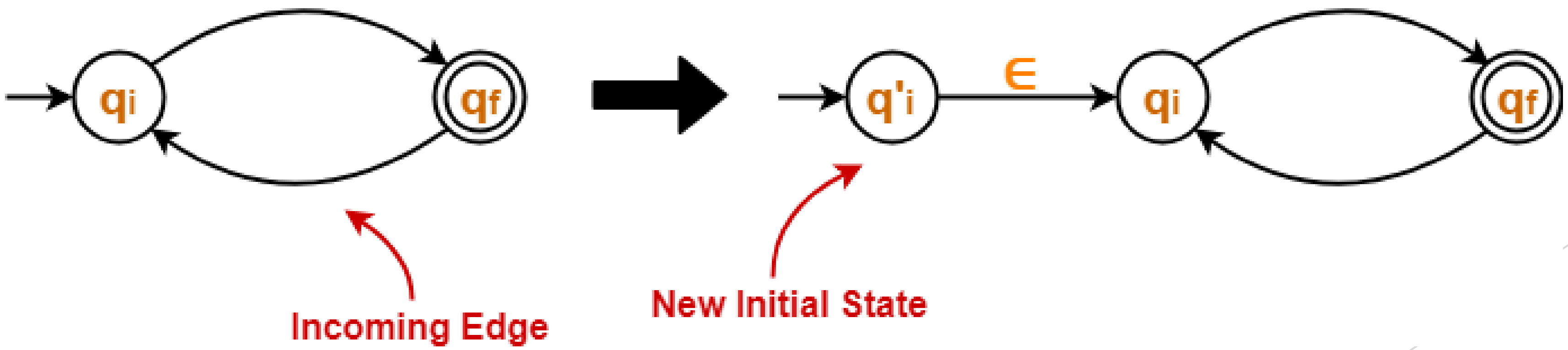
GNFA state reduction



From any GNFA, we can eliminate every state but the start and accept states

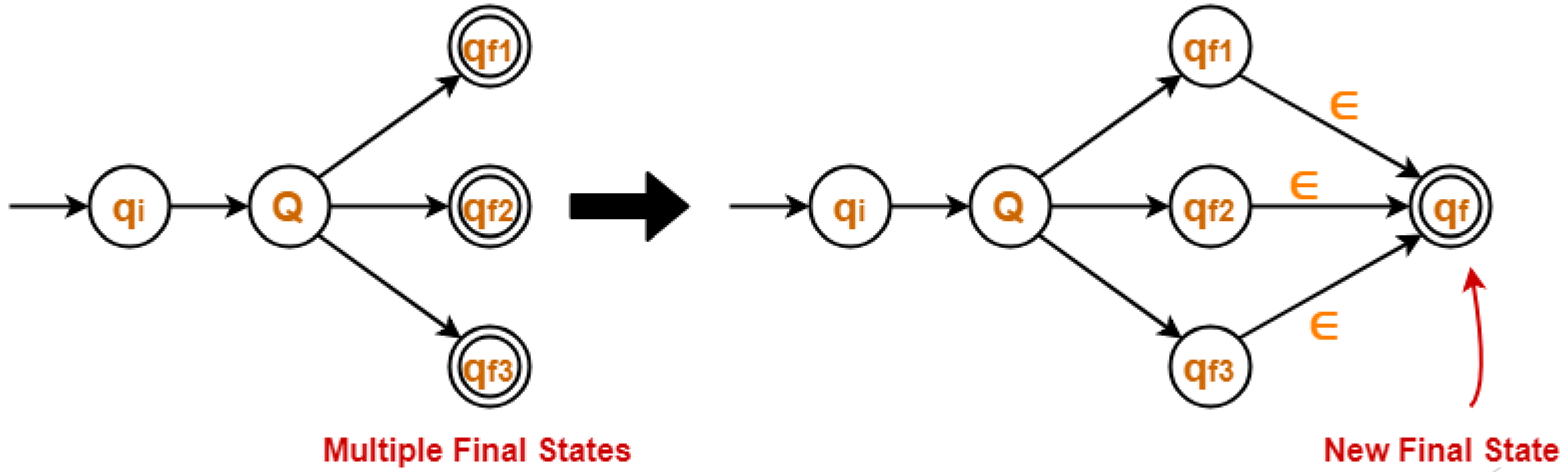
State Elimination Method

- **Step 1:** The initial state of the DFA must not have any incoming edge.
- If there exists any incoming edge to the initial state, then create a new initial state having no incoming edge to it.



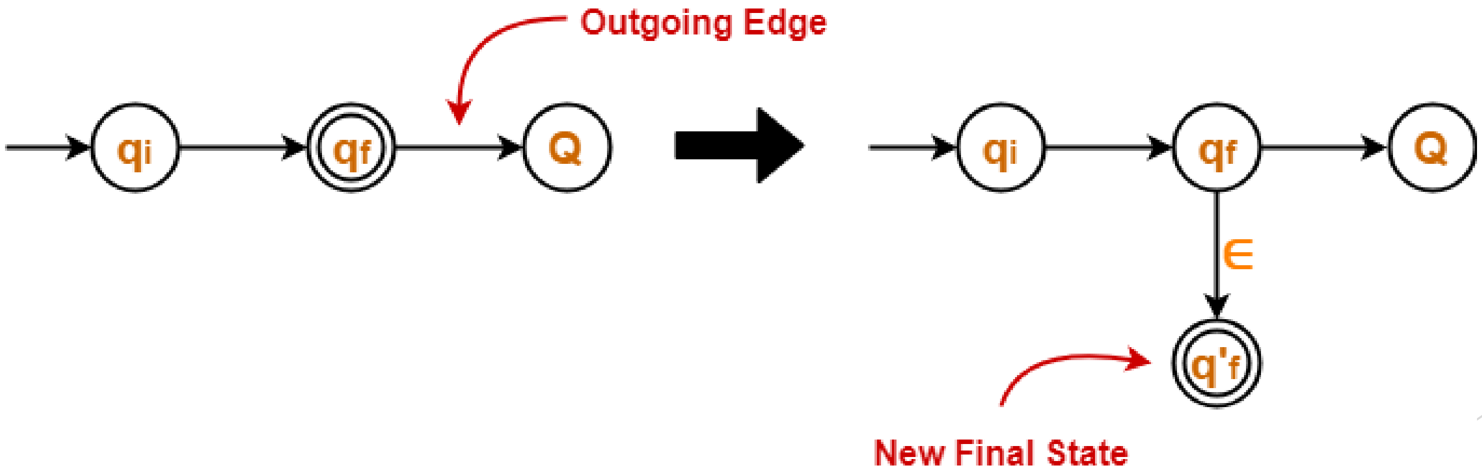
Continue . . .

- **Step 2:** There must exist only one final state in the DFA.
- If there exists multiple final states in the DFA, then convert all the final states into non-final states and create a new single final state.



Continue . . .

- **Step 3:** The final state of the DFA must not have any outgoing edge.
- If there exists any outgoing edge from the final state, then create a new final state having no outgoing edge from it.



Continue . . .

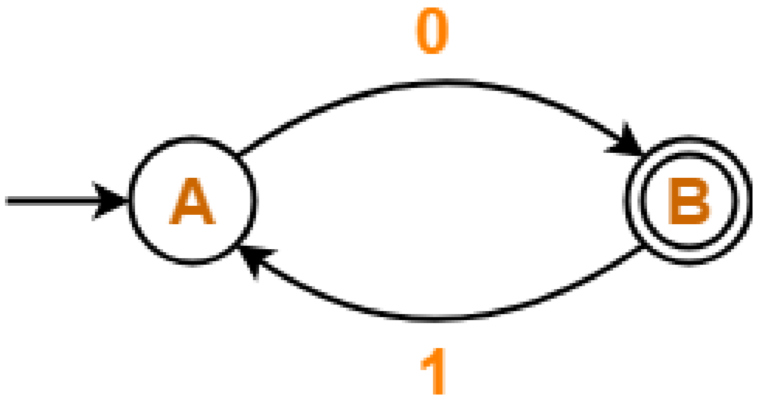
- **Step 4:** Eliminate all the intermediate states one by one.
- These states may be eliminated in any order.
- In the end,
  - Only an initial state going to the final state will be left.
  - The cost of this transition is the required regular expression.

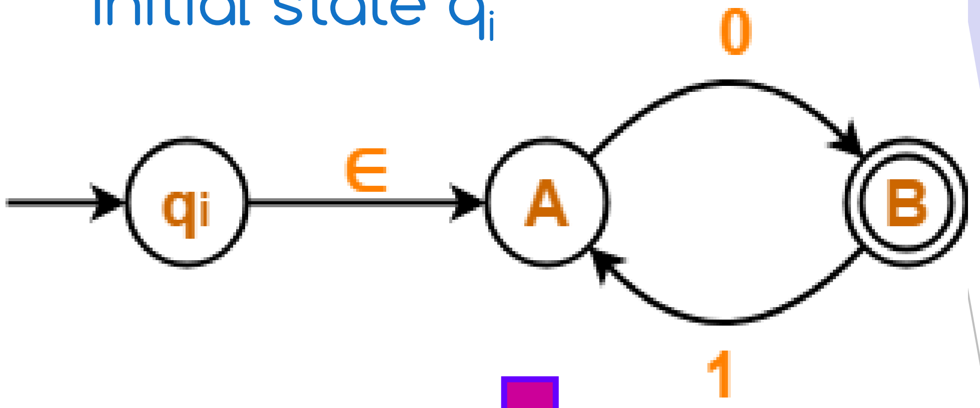
NOTE

- The state elimination method can be applied to any finite automata.

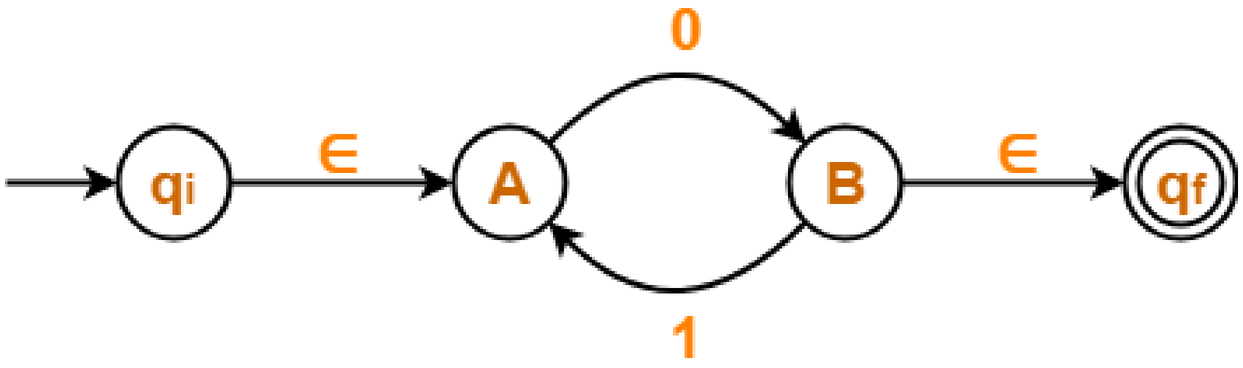
(NFA, $\in$-NFA, DFA etc)

Example 1



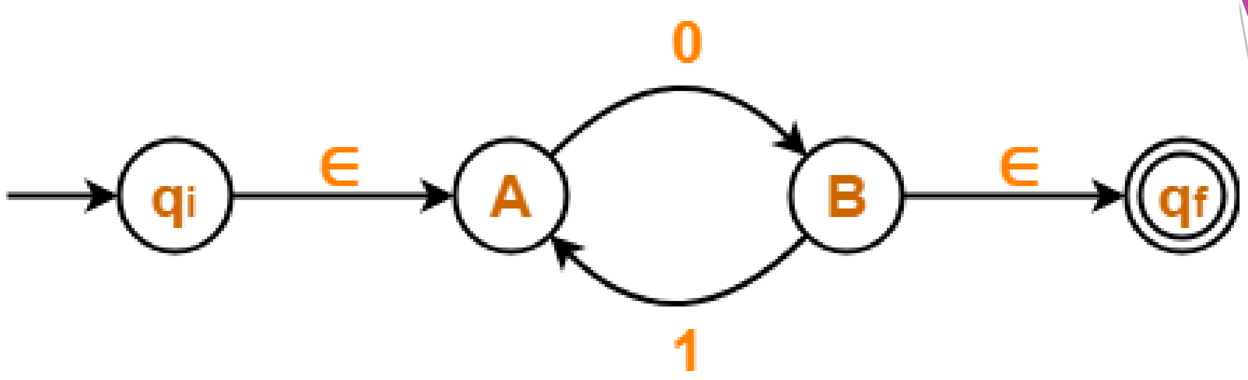


1. create a new initial state q_i



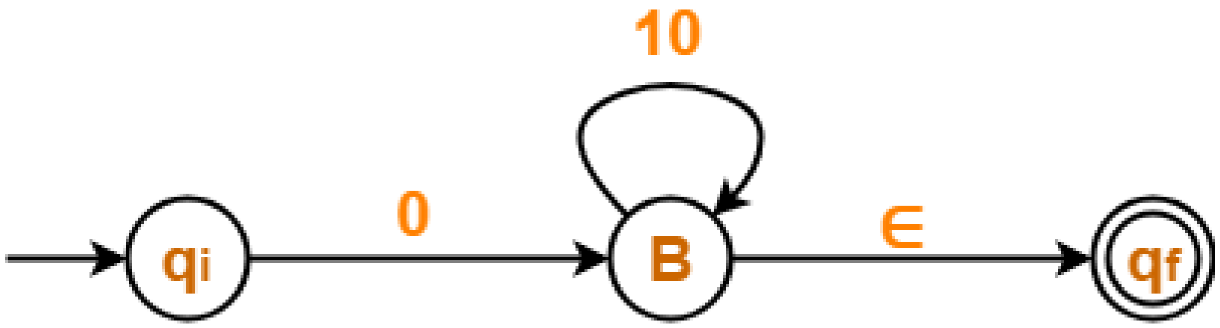
2. create a new final state q_f

Example 1



Example 1

Eliminate state A



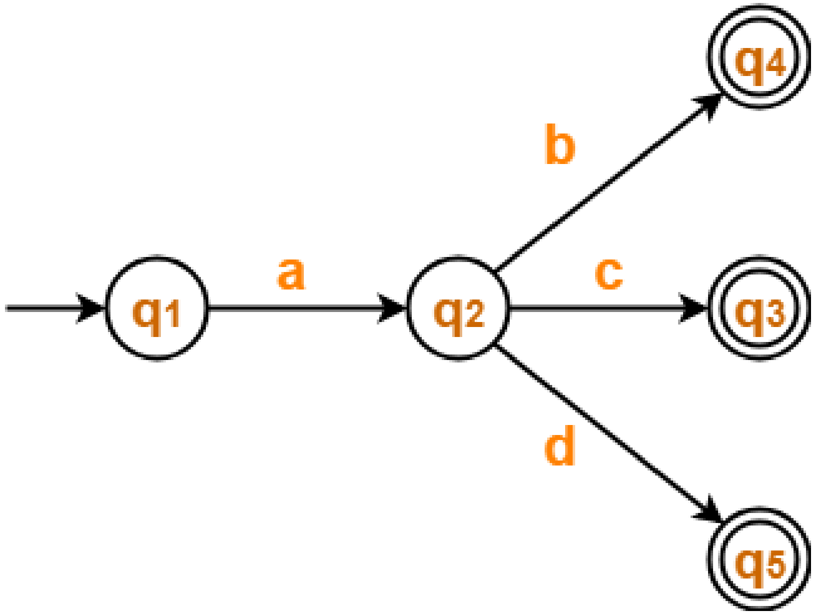
Eliminate state B

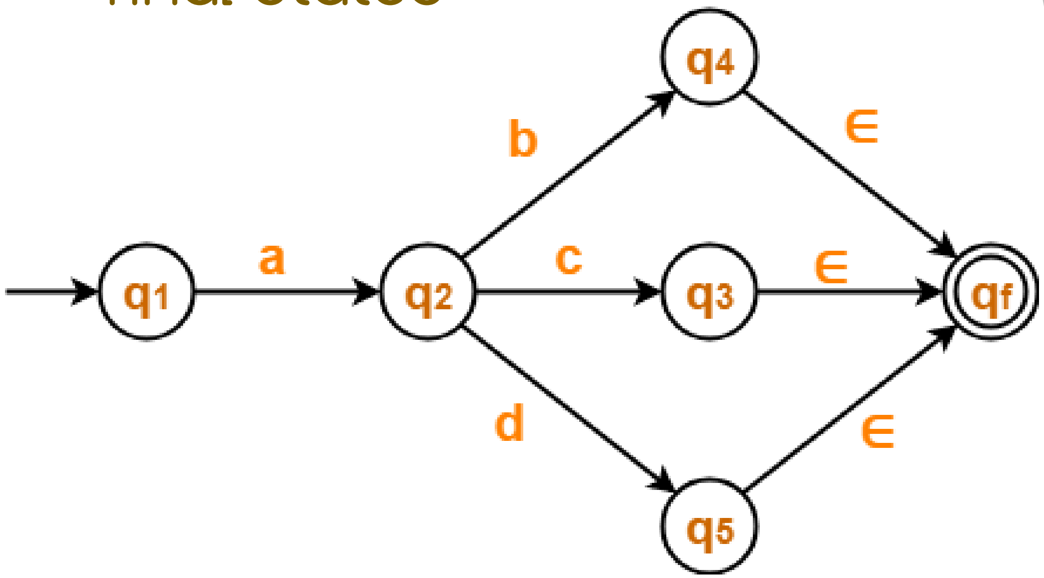
Regular Expression = 0(10)*

If we first eliminate state B and then state A, then regular expression would be = (01)*0. This is also the same and correct.

Example 2

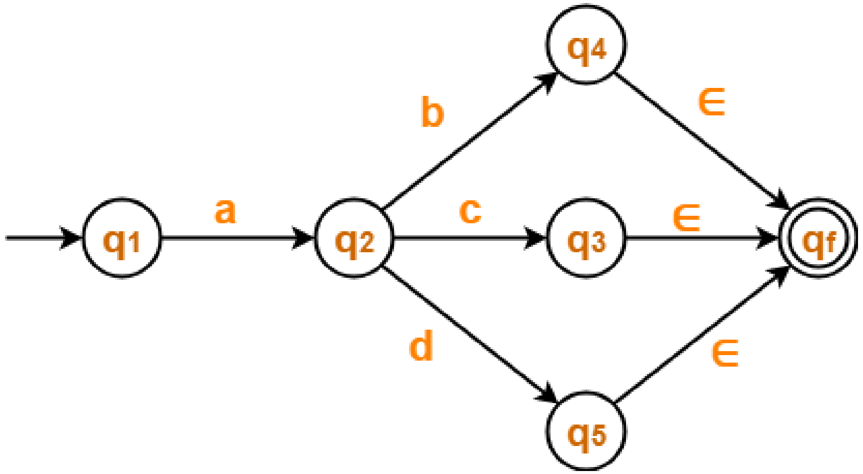
There exists multiple final states



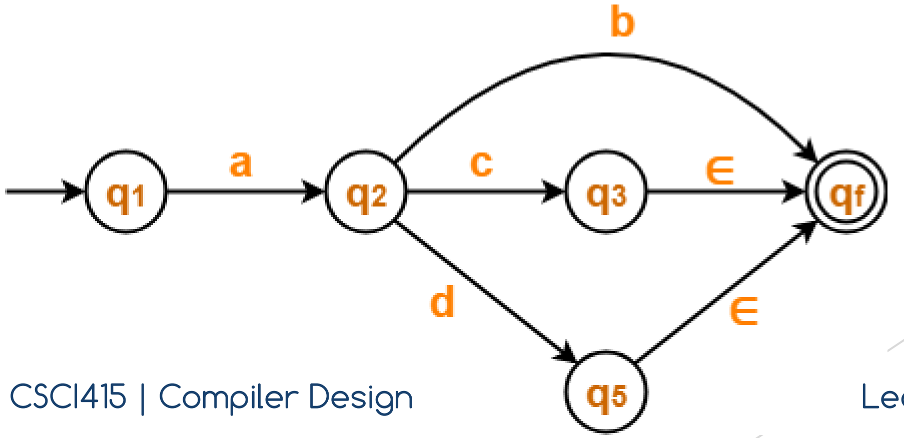


Example 2

Start eliminating the intermediate states

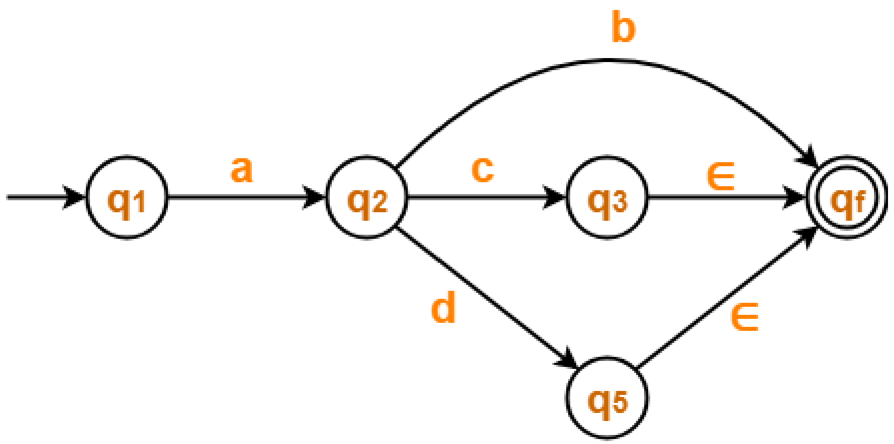


- First, let us eliminate state q_4.
- There is a path going from state q_2 to state q_f via state q_4.
- So, after eliminating state q_4 , we put a direct path from state q_2 to state q_f having cost $b.\epsilon = b$.

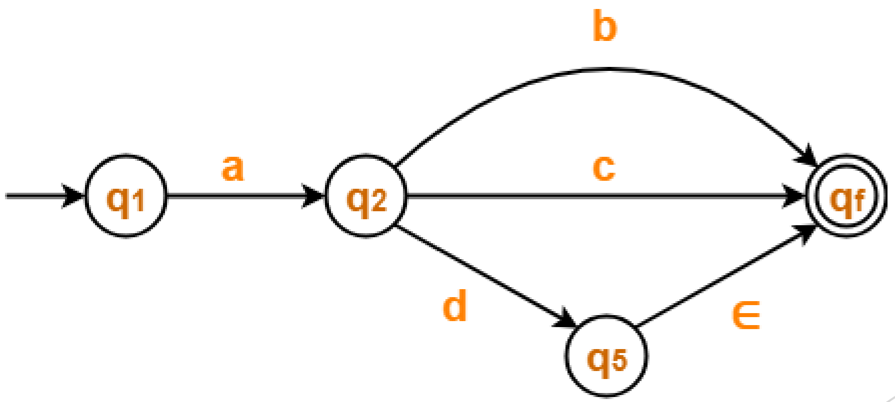


Example 2

Eliminate state q3

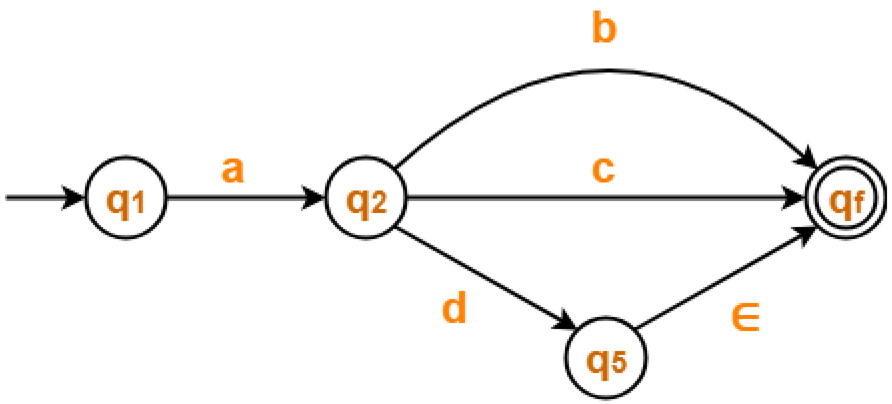


- There is a path going from state q2 to state qf via state q3.
- So, after eliminating state q3 , we put a direct path from state q2 to state qf having cost c.∈ = c

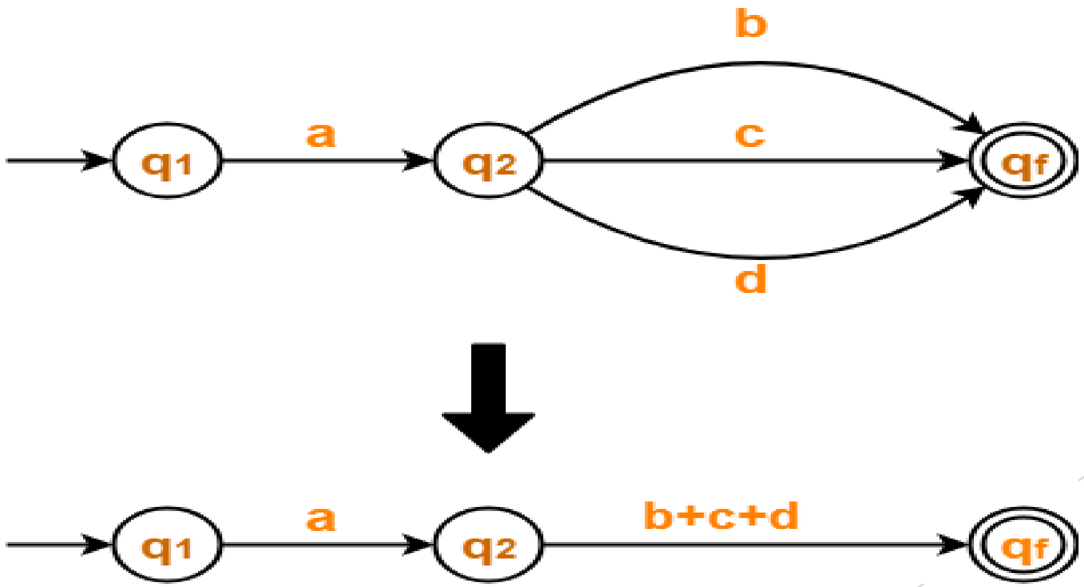


Example 2

Eliminate state q5



- There is a path going from state q2 to state qf via state q5.
- So, after eliminating state q5 , we put a direct path from state q2 to state qf having cost d.∈ = d.



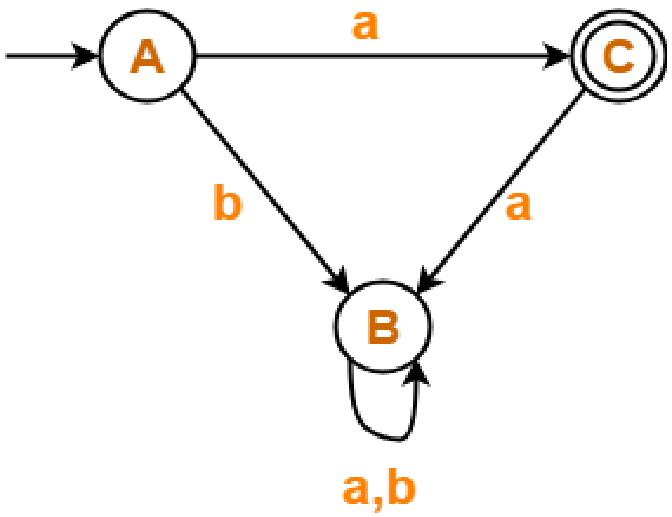
Example 2

Eliminate state q2

- There is a path going from state q1 to state qf via state q2.
- So, after eliminating state q2 , we put a direct path from state q1 to state qf having cost a.(b+c+d).

Regular Expression = a(b+c+d)

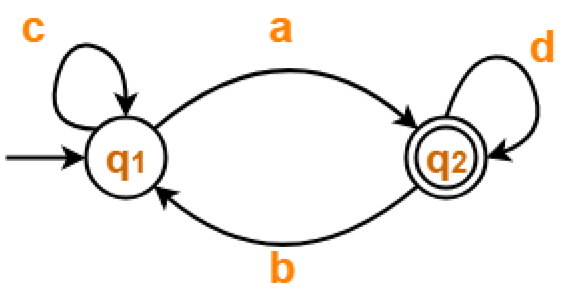
Example 3



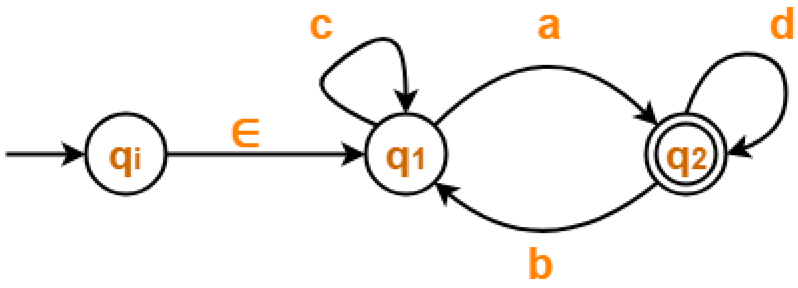
- State B is a dead state as it does not reach to the final state.
- So, we eliminate state B and its associated edges.

Regular Expression = a

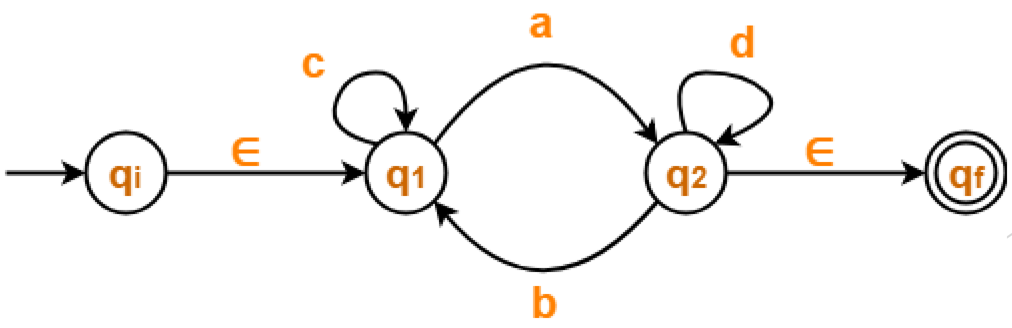
Example 4



Create a new initial state q_i

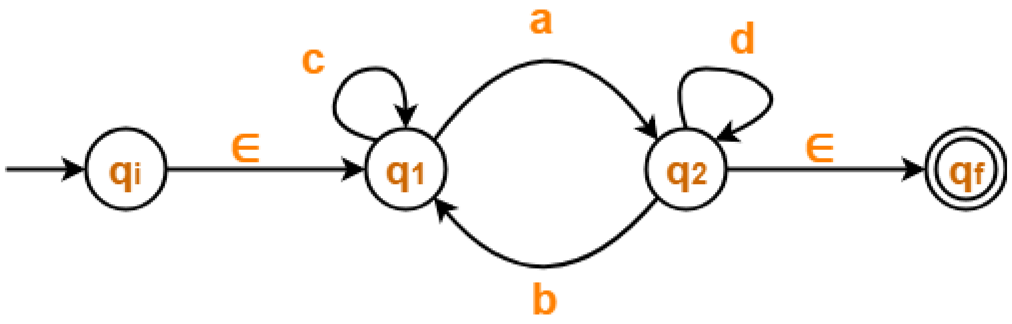


Create a new final state q_f

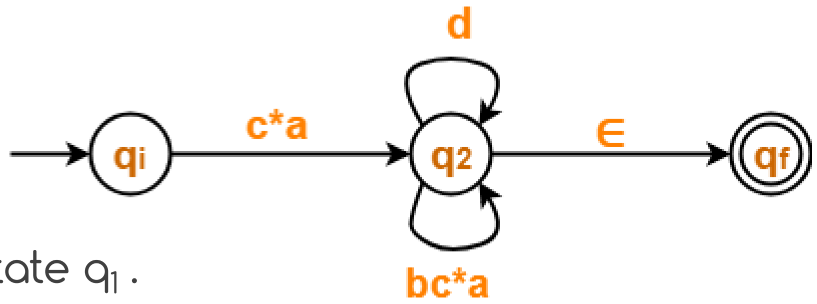


Example 4

Start eliminating the intermediate



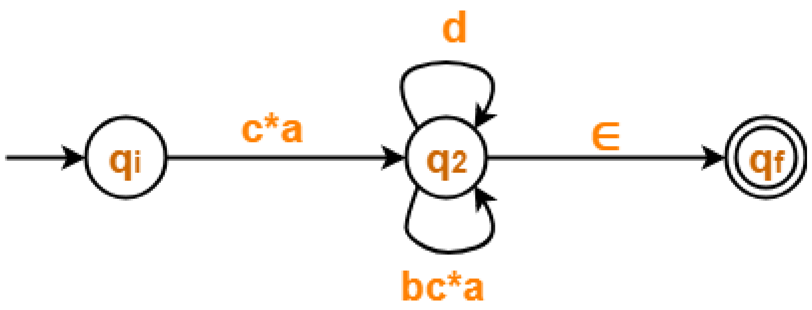
Eliminate State q1



- There is a path going from state q_i to state q_2 via state q_1 .
- Put a direct path from state q_i to state q_2 having cost ∈.c*.a = c*a
- There is a loop on state q_2 using state q_1 .
- So, after eliminating state q_1 , put a direct loop on state q_2 having cost b.c*.a = bc*a

Example 4

Eliminate State q2



- There is a path going from state q_i to state q_f via state q_2 .
- So, after eliminating state q_2, we put a direct path from state q_i to state q_f having cost c*a(d+bc*a)*∈ = c*a(d+bc*a)*

Regular Expression = c*a(d+bc*a)*

Summary of Lexical Analysis

- RE to NFA: Thompson's construction
  - Core insight: inductively build up NFA using “templates”
  - Core concept: use null transitions to build NFA quickly
- NFA to DFA: Subset construction
  - Core insight: DFA nodes represent subsets of NFA nodes
  - Core concept: use null closure to calculate subsets
- DFA minimization: Equivalence Theorem
  - Core insight: create partitions, then keep splitting
- DFA to RE: State Elimination Method
  - repeatedly eliminate states by combining regexes

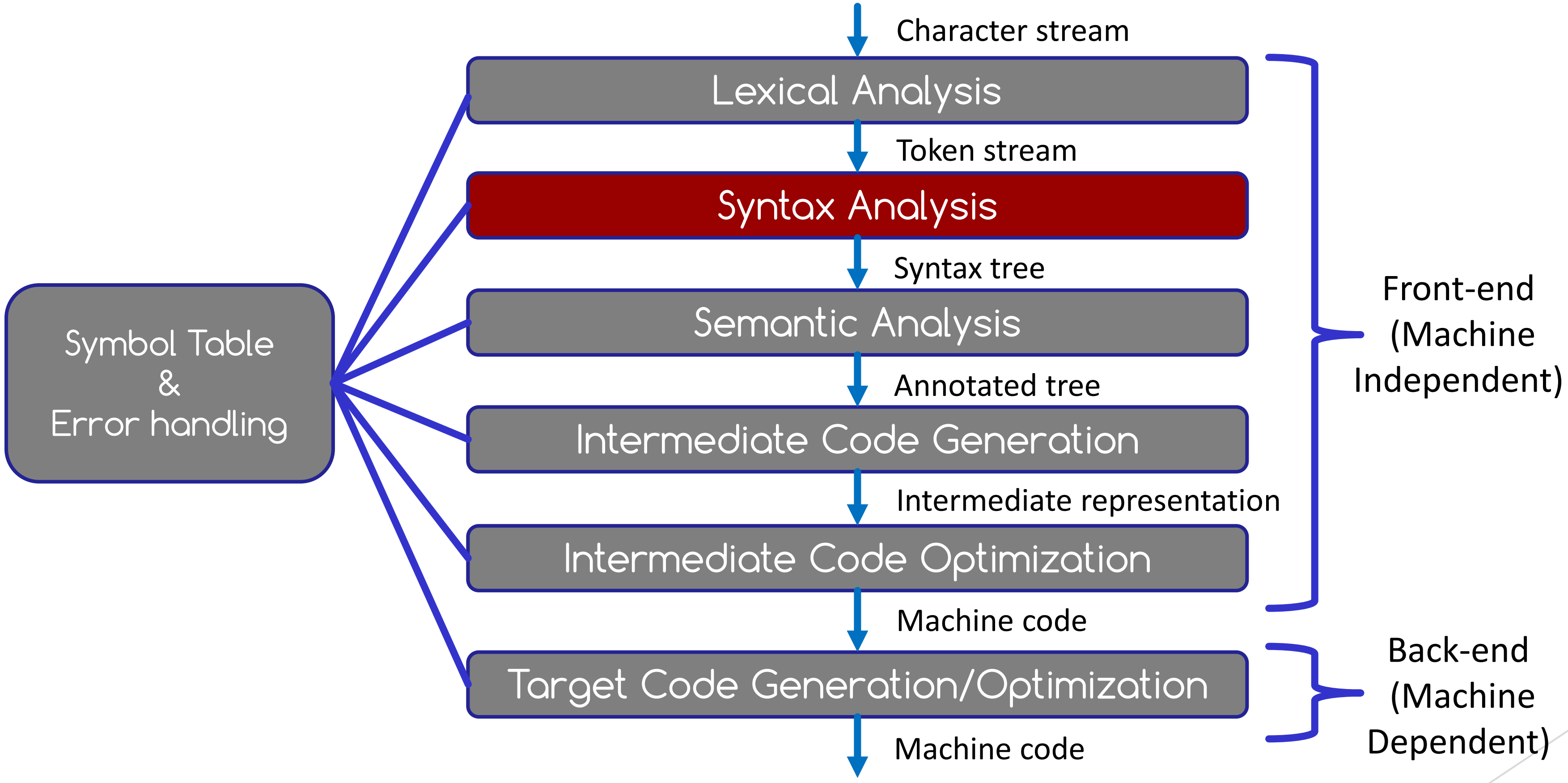
The Cycle of Constructions


Q
A

2 Syntax Analysis

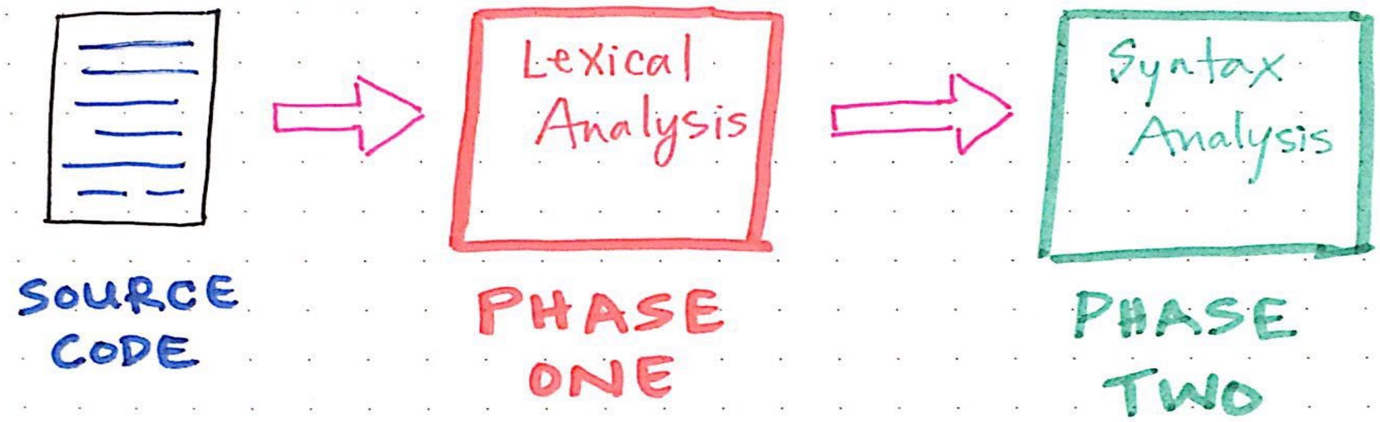
- The Parsing process
- Context free grammar (CFG)

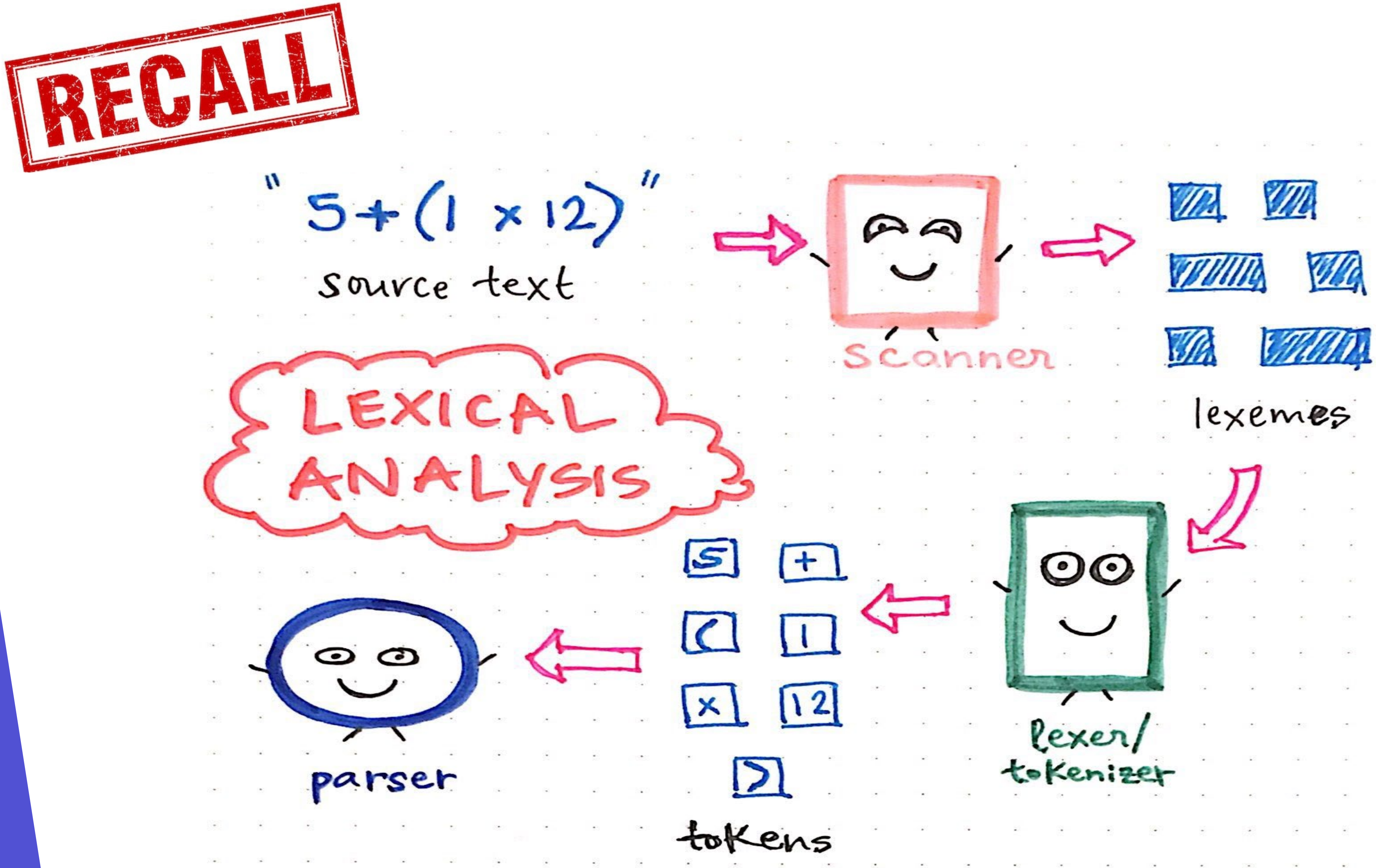
Compiler Phases

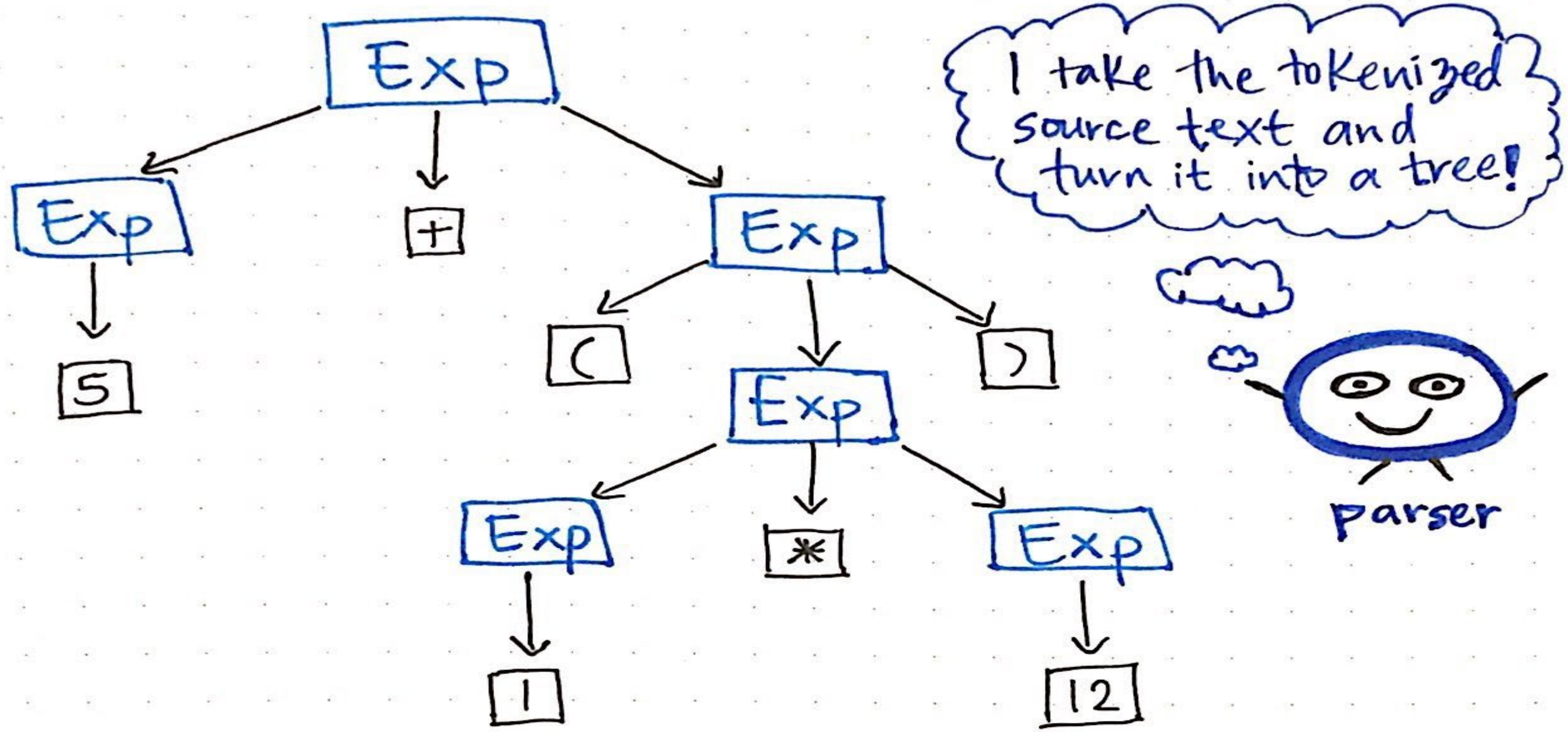


RECALL

⇒ Before any code from a source program, written in any language, can be parsed, it must first be scanned, split up, and grouped in certain ways. This is the first phase of the compilation process, called lexical analysis.



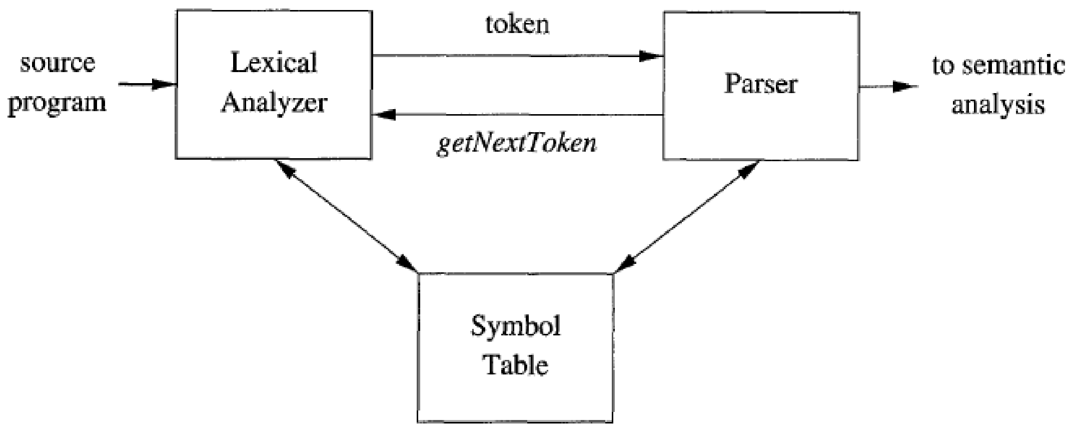

RECALL
"5+(1 x 12)"
source text
scanner
lexemes
LEXICAL ANALYSIS
lexer/
tokenizer
5
+
(
1
x
12
)
tokens
parser



* First comes the lexical analysis phase, followed by the syntax analysis phase, which will generate a parse tree.

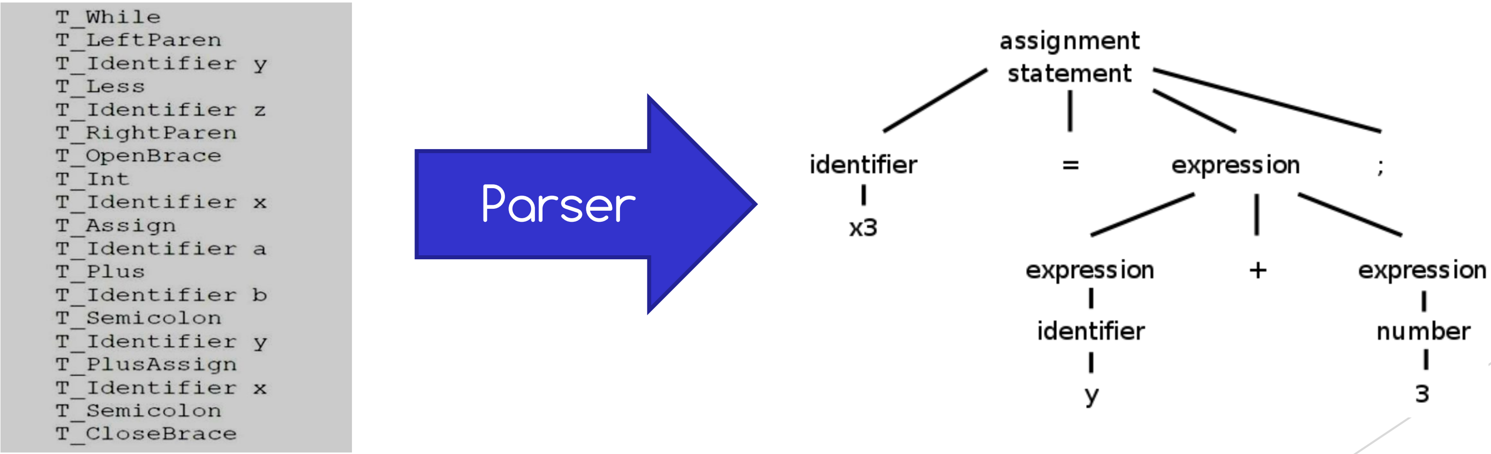
Syntax Analysis

- Syntax analysis is all about discovering **structure in code**. It determines whether or not a text follows the expected format.
- The main aim of this phase is to make sure that the **source code** was written by the programmer **is correct or not**.
- It also determines the structure of source language and **grammar** or syntax of the language.



Syntax Analysis

- Syntax analyzer is responsible for performing the following tasks:
  - Obtain tokens from the lexical analyzer
  - Checks if the expression is syntactically correct or not
  - Report all syntax errors
  - Construct a hierarchical structure which is known as a parse tree



Sequence of tokens

Syntax Tree

Lexical versus Syntax Analysis

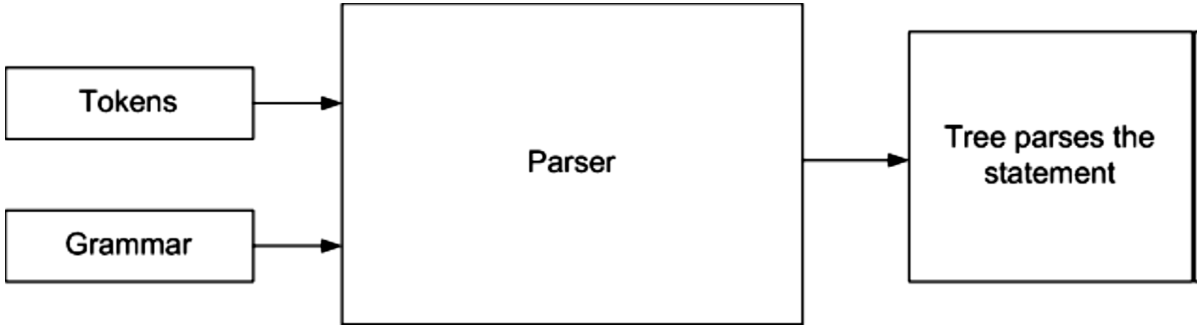
Why separate lexical analysis from parsing?

- Simplicity of design
  - simplify both the lexical analysis and the syntax analysis.
- Efficiency
  - specialized techniques can be applied to improve lexical analysis.
- Portability
  - only the scanner needs to communicate with the outside.

1 The Parsing Process

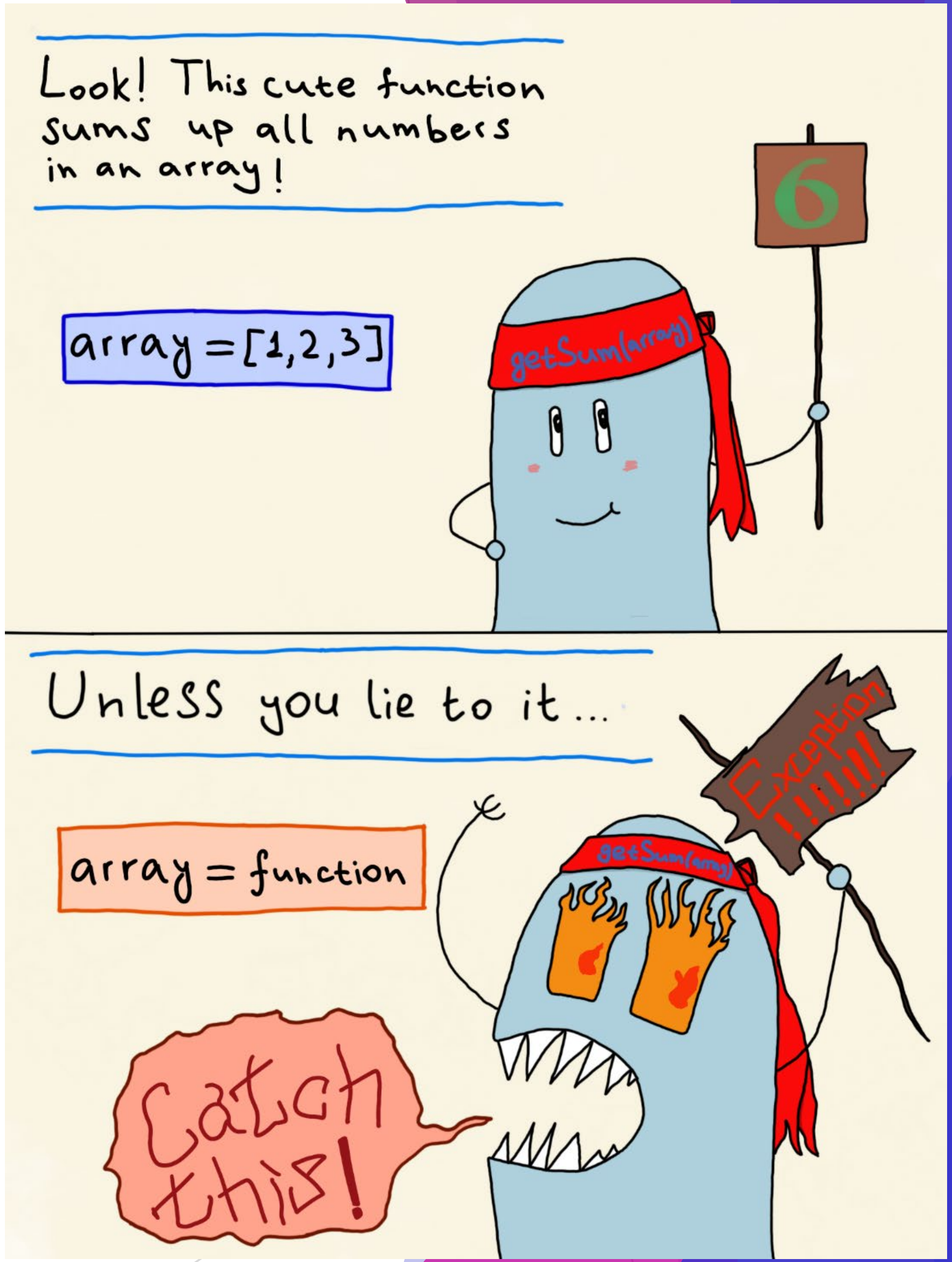
Introduction

- Parsing is the task of **Syntax Analysis**
  - Determining the syntax, or structure, of a program.
- The syntax is defined by the **grammar rules of a Context-Free Grammar**
  - The rules of a context-free grammar are **recursive**
- The basic data structure of Syntax Analysis is **parse tree** or **syntax tree**
  - The syntactic structure of a language must also be recursive

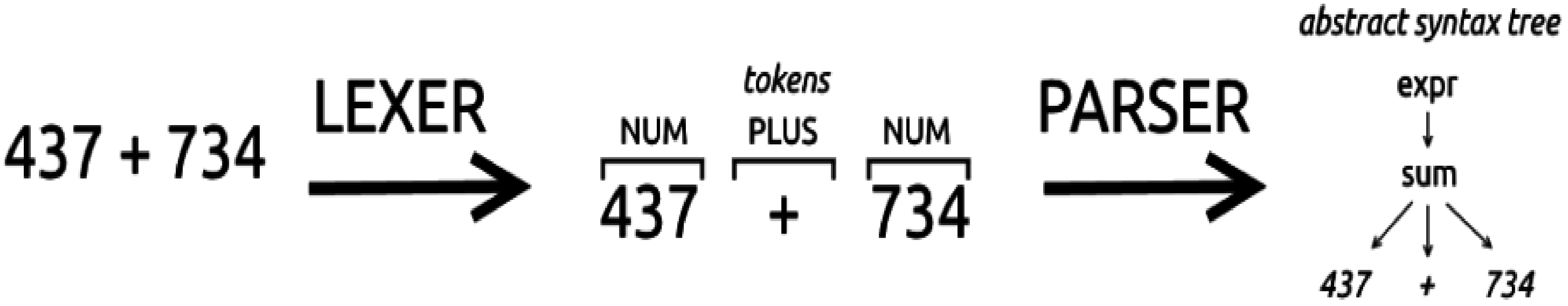


Goals

1. Construct the **structure** described by that series of tokens. (Parse/Syntax Tree Construction)
2. Report **errors** if those tokens do not properly encode a structure.



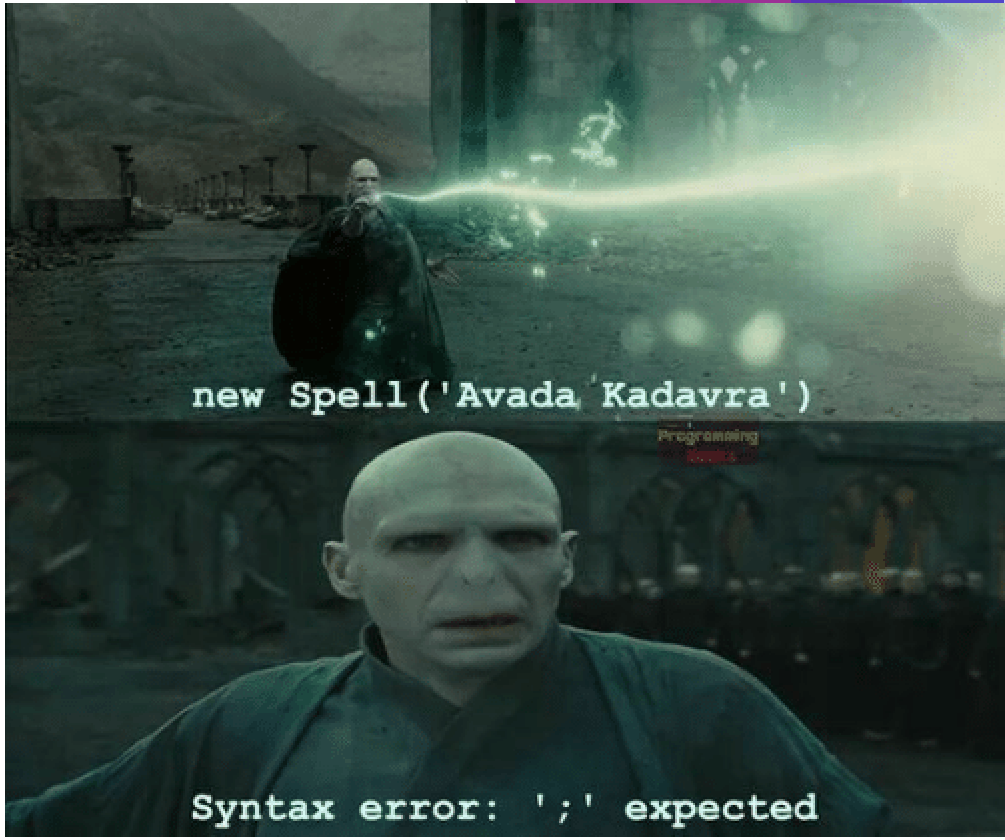
Function of a Parser



- Takes the sequence of tokens produced by the scanner as its input and produces the syntax tree as its output.

Issues of the Parsing

- Error in the parser
  - The parser must not only report an error message
  - **but it must recover from the error and continue parsing** (to find as many errors as possible)
- A parser may perform **error repair**
  - Error recovery is the reporting of meaningful error messages and the resumption of parsing as close to the actual error as possible



Compilers will not forgive..

2 Context-Free Grammars

Context-Free Grammars

- A context-free grammar (or CFG) is a formalism for defining languages.
- Can define the context-free languages, a strict superset of the regular languages.
  - Similar to the specification of the lexical structure of a language using regular expressions
- For example:

$$\text{exp} \rightarrow \text{exp op exp} \mid (\text{exp}) \mid \textbf{number}$$

$$\text{op} \rightarrow + \mid - \mid *$$

Context-Free Grammars

Formally, a context-free grammar is a collection of four objects:

1. A set of **nonterminal symbols** (or variables):
   - Capital Letters(A,B,..etc)
2. A set of **terminal symbols**:
   - Lower-Case Letters (a,b,t,u,v…etc)
3. A set of **production rules** saying how each nonterminal can be converted by a string of terminals and nonterminals: →
4. A **start symbol** that begins the derivation.

- Also Lowercase **Greek** letters are used to represent **arbitrary strings of terminals and nonterminals. i.e.** α, γ, ω

E → int

E → E Op E

E → (E)

Op → +

Op → -

Op → *

Op → /

CFG Rules

- Grammar rules use **regular expressions** as **components**
- The notation was developed by **John Backus** and adapted by **Peter Naur**
- Grammar rules in this form are usually said to be in **B**ackus-**N**aur **F**orm, or **BNF**

Comparison to Regular Expression Notation

Comparing an Example

- The context-free grammar:

  exp → exp op exp | (exp) | **number**

  op → + | – | *

- The regular expression:

  number = digit digit*

  digit = 0|1|2|3|4|5|6|7|8|9

Comparison between CFG and RE

Regular Expression Rules	CFG Rules
Three operations: • Choice • Concatenation • Repetition	• Vertical bar appears as meta-symbol for choice. • Concatenation is used as a standard operation. • No meta-symbol for repetition (like the * of regular expressions)
Equal sign represents the definition of a name for a regular expression	Use the arrow symbol → instead of equality to express the definitions of names
Name is written in italics to distinguish it from a sequence of actual characters.	Names (Non-Terminals) are written in italic (in a different font)

Construction of a CFG rule

- A grammar rule in BNF is interpreted as follows:
  - The rule defines the structure whose name is to the left of the arrow
  - The structure is defined to consist of one of the choices on the right-hand side separated by the vertical bars
  - The **sequences** of symbols and structure names within each choice defines the layout of the structure
  - For example:
    - $exp \rightarrow exp\ op\ exp \mid (exp) \mid \textbf{number}$
    - $op \rightarrow + \mid - \mid *$

Example 1
Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

$\mathbf{E} \rightarrow \texttt{int}$

$\mathbf{E} \rightarrow \mathbf{E\ Op\ E}$

$\mathbf{E} \rightarrow \texttt{(}\mathbf{E}\texttt{)}$

$\mathbf{Op} \rightarrow \texttt{+}$

$\mathbf{Op} \rightarrow \texttt{-}$

$\mathbf{Op} \rightarrow \texttt{*}$

$\mathbf{Op} \rightarrow \texttt{/}$

Example 1
Arithmetic Expressions

(a) How to generate /derive: **int * (int +int)**

$$
\begin{aligned}
E &\rightarrow \texttt{int} \\
E &\rightarrow E\ Op\ E \\
E &\rightarrow \texttt{(}E\texttt{)} \\
Op &\rightarrow \texttt{+} \\
Op &\rightarrow \texttt{-} \\
Op &\rightarrow \texttt{*} \\
Op &\rightarrow \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
& E \\
\Rightarrow\ & E\ Op\ E \\
\Rightarrow\ & E\ Op\ \texttt{(}E\texttt{)} \\
\Rightarrow\ & E\ Op\ \texttt{(}E\ Op\ E\texttt{)} \\
\Rightarrow\ & E\ \texttt{*}\ \texttt{(}E\ Op\ E\texttt{)} \\
\Rightarrow\ & \texttt{int * (}E\ Op\ E\texttt{)} \\
\Rightarrow\ & \texttt{int * (int } Op\ E\texttt{)} \\
\Rightarrow\ & \texttt{int * (int } Op \texttt{ int)} \\
\Rightarrow\ & \texttt{int * (int + int)}
\end{aligned}
$$

Example 1
Arithmetic Expressions

(b) How to generate /derive: **int / int** ?

$E \rightarrow \texttt{int}$

$E \rightarrow E\ Op\ E$

$E \rightarrow \texttt{(}E\texttt{)}$

$Op \rightarrow \texttt{+}$

$Op \rightarrow \texttt{-}$

$Op \rightarrow \texttt{*}$

$Op \rightarrow \texttt{/}$

$$
\begin{aligned}
& E \\
\Rightarrow\ & E\ Op\ E \\
\Rightarrow\ & E\ Op\ \texttt{int} \\
\Rightarrow\ & \texttt{int}\ Op\ \texttt{int} \\
\Rightarrow\ & \texttt{int / int}
\end{aligned}
$$

Example 1
Arithmetic Expressions

A Notational Shorthand

$E \rightarrow \texttt{int}$

$E \rightarrow E\ Op\ E$

$E \rightarrow \texttt{(}E\texttt{)}$

$Op \rightarrow \texttt{+}$

$Op \rightarrow \texttt{-}$

$Op \rightarrow \texttt{*}$

$Op \rightarrow \texttt{/}$

$$E \rightarrow \texttt{int} \mid E\ Op\ E \mid \texttt{(}E\texttt{)}$$

$$Op \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$$

How Grammar Determine a Language

Context-free grammar rules determine the set of syntactically legal strings of token symbols for the structures defined by the rules.

- For example, the arithmetic expression (34-3)*42
- Corresponds to the legal string of seven tokens
  - (number - number) * number
- While (34-3*42 is not a legal expression,
  - There is a left parenthesis that is not matched by a right parenthesis and the second choice in the grammar rule for an *exp* requires that parentheses be generated in pairs

Recursion in CFG

Recursion

- The grammar rule:
  - $A \rightarrow A\,a \mid a$ or $A \rightarrow a\,A \mid a$
  - Generates the language $\{a_n \mid n$ an integer $>=1\} = L(a+)$

    (the set of all strings of one or more a's)
  - The same language as that generated by the regular expression a+
- The string *aaaa* can be generated by the first grammar rule with the derivation
  - $A \Rightarrow Aa \Rightarrow Aaa \Rightarrow Aaaa \Rightarrow aaaa$

Recursion

- To generate the same language as the regular expression a* we must have a notation for a grammar rule that generates the empty string
  - use the epsilon meta-symbol for the empty string
  - *empty* $\rightarrow$ ε, called an ε-production (an "epsilon production").
- A grammar that generates a language containing the empty string must have at least one ε-production.

Example 1
CFG Recursion a*

- A grammar equivalent to the regular expression a*
  - $A \rightarrow A\,a \mid \varepsilon$ or $A \rightarrow a\,A \mid \varepsilon$

- Both grammars generate the language
  - $\{ a^n \mid n \text{ an integer} >= 0\} = L(a^*)$.

Recursion

- left recursive:
  - The non-terminal *A* appears as the first symbol on the left-hand side of the rule defining *A*
  - $A \rightarrow A\, a \mid a$
- right recursive:
  - The non-terminal *A* appears as the last symbol on the right-hand side of the rule defining *A*
  - $A \rightarrow a\, A \mid a$

Example 2

- Consider a rule of the form: $A \rightarrow A\alpha \mid \beta$
  - where α and β represent arbitrary strings and β does not begin with A.
- This rule generates all strings of the form
  - $\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots$
  - (all strings beginning with a β, followed by 0 or more α's).
- This grammar rule is **equivalent in its effect to the regular expression $\beta\alpha^*$.**
- Similarly, the right recursive grammar rule $A \rightarrow \alpha\, A \mid \beta$
  - (where β does not end in A)
  - generates all strings $\beta, \alpha\beta, \alpha\alpha\beta, \alpha\alpha\alpha\beta, \ldots.$

Example 3

- The if-statement grammar can be written in the following way using an ε-production:

  statement → *if-stmt* | *other*

  if-stmt → if (*exp*) *statement else-part*

  else-part → else *statement* | ε

  exp → 0 | 1

- The ε-production indicates that the structure *else-part* is optional.

Example 4

- Consider the following grammar *G* for a sequence of statements:

  stmt-sequence $\rightarrow$ *stmt* ; *stmt-sequence* / *stmt*

  stmt $\rightarrow$ s

- This grammar generates sequences of one or more statements separated by semicolons
  - (statements have been abstracted into the single terminal s):
- *L*(*G*)= { s, s; s, s; s; s,...)

Example 6
Language of palindromes

- Let Σ = {a, b} and let $L = \{ w \in \Sigma^* \mid w$ is a palindrome $\}$
- We can design a CFG for L by thinking inductively:
- Base case: ε, a, and b are palindromes.
- If ω is a palindrome, then aωa and bωb are palindromes.

S → ε
S → a
S → b
S → aSa
S → bSb

Inductive definition

A Notational Shorthand

S → ε | a | b | aSa | bSb

CFG Caveats

- Is the following grammar a CFG for the language $\{ a^n b^n \mid n \in \mathbb{N} \}$?

$$S \rightarrow aSb$$

- What strings can you derive?
  - Answer: None!
- What is the language of the grammar?
  - Answer: φ
- When designing CFGs, make sure your recursion actually terminates!

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

$$S \rightarrow a^*b$$

$$\Downarrow$$

$$S \rightarrow Ab$$

$$A \rightarrow Aa \mid \varepsilon$$

From Regexes to CFGs

- CFGs don't have the Kleene star, parenthesized expressions, or internal | operators.
- However, we can convert regular expressions to CFGs as follows:

$S \rightarrow a(b \mid c^*)$

$$S \rightarrow aX$$
$$X \rightarrow (b \mid c^*)$$

$$S \rightarrow aX$$
$$X \rightarrow b \mid C$$

$$S \rightarrow aX$$
$$X \rightarrow b \mid C$$
$$C \rightarrow Cc \mid \varepsilon$$

The Language of a Grammar

- Consider the following CFG G

$$S \rightarrow aSb \mid \varepsilon$$

- What strings can this generate?

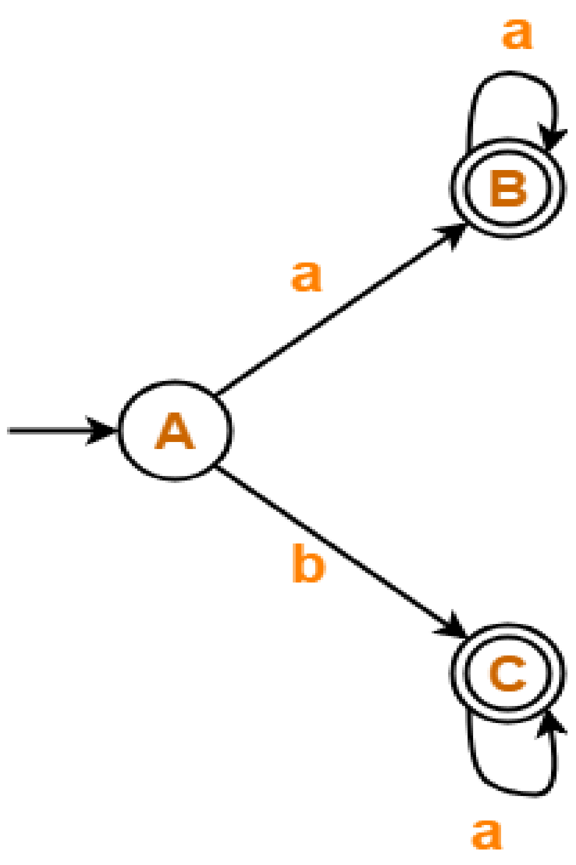
a	a	a	a	a	a	b	b	b	b	b	b

$$\mathscr{L}(G) = \{ a^n b^n \mid n \in \mathbb{N} \}$$

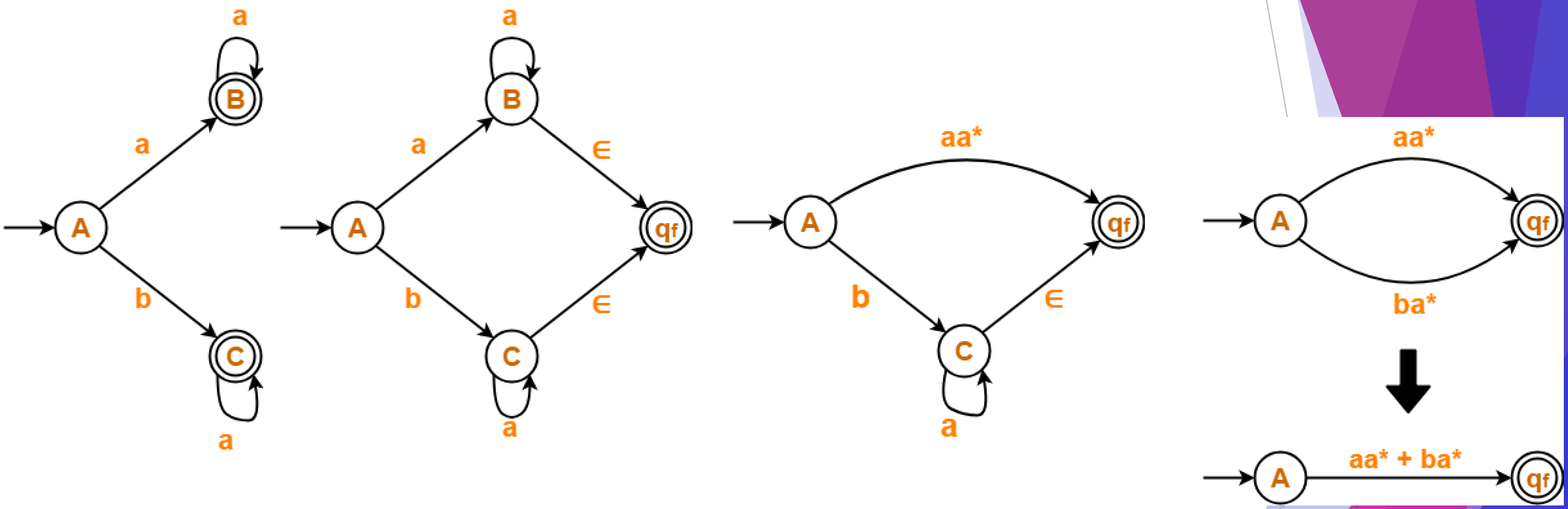

Q
A

Review Questions

DFA to RE: Question 1



DFA to RE: Question 1 Solution



Regular Expression = aa* + ba*

Problem 1

- Let $\Sigma = \{l, r\}$ and let $L = \{w \in \Sigma^* \mid w$ has the same number of l's and r's $\}$
- Is this a grammar for L?

$$S \rightarrow lSr \mid rSl \mid \varepsilon$$

Can you derive the string lrrl?

Problem 2: Derivation of Chemicals

Form → Cmp | Cmp Ion

Cmp → Term | Term Num | Cmp Cmp

Term → Elem | (Cmp)

Elem → H | He | Li | Be | B | C | …

Ion → + | - | IonNum + | IonNum -

IonNum → 2 | 3 | 4 | ...

Num → 1 | IonNum

$C_{19}H_{14}O_5S$

$Cu_3(CO_3)_2(OH)_2$

MnO_4^-

S^{2-}

Problem 2: Derivation of Chemicals

Form → Cmp | Cmp Ion

Cmp → Term | Term Num | Cmp Cmp

Term → Elem | (Cmp)

Elem → H | He | Li | Be | B | C | …

Ion → + | - | IonNum + | IonNum -

IonNum → 2 | 3 | 4 | …

Num → 1 | IonNum

MnO_4^-

Form

⇒ Cmp Ion

⇒ Cmp Cmp Ion

⇒ Cmp Term Num Ion

⇒ Term Term Num Ion

⇒ Elem Term Num Ion

⇒ Mn Term Num Ion

⇒ Mn Elem Num Ion

⇒ MnO Num Ion

⇒ MnO IonNum Ion

⇒ MnO_4 Ion

⇒ MnO_4^-

Problem 2: Derivation of Chemicals

Form → Cmp | Cmp Ion

Cmp → Term | Term Num | Cmp Cmp

Term → Elem | (Cmp)

Elem → H | He | Li | Be | B | C | …

Ion → + | - | IonNum + | IonNum -

IonNum → 2 | 3 | 4 | …

Num → 1 | IonNum

MnO_4^-

Form
⇒ Cmp Ion
⇒ Cmp Cmp Ion
⇒ Cmp Term Num Ion
⇒ Term Term Num Ion
⇒ Elem Term Num Ion
⇒ Mn Term Num Ion
⇒ Mn Elem Num Ion
⇒ MnO Num Ion
⇒ MnO IonNum Ion
⇒ MnO_4 Ion
⇒ MnO_4^-

Problem 3

$$A \rightarrow (A)\,A \;/\; \varepsilon$$

- generates the strings of all "balanced parentheses."
- For example, the string (( ) (( ))) ( )
  - generated by the following derivation
  - (the ε-production is used to make A disappear as needed):
  - A => (A) A => (A)(A)A => (A)(A) =>(A)( ) => ((A)A)( ) =>(( )A)() => (( ) (A)A) () => (( )(A))( ) => (( )((A)A))( ) => (( )(( )A))( ) => (( )(( )))( )

Problem 4

- Write the CFG that can generate the following expression

$$x= -c + a/b$$

References of this lecture

- Presentation slides of the book: COMPILER CONSTRUCTION, Principles and Practice, by Kenneth C. Louden
- Credits for Dr. Sally Saad, Prof. Mostafa Aref, Dr. Islam Hegazy, and Dr. Abd ElAziz for help in content preparation and aggregation (FCIS-ASU)

Quote of the Day ☺

Be like Compiler...

.

.

Learn to ignore the nonsense comments. 😒

#coder #programmingquotes

6:10 pm · 18 Jun 19 · Twitter for Android

See you next lecture