Compiler Construction

Lecture 2: Lexical Analysis I (Simple Matching Problem)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

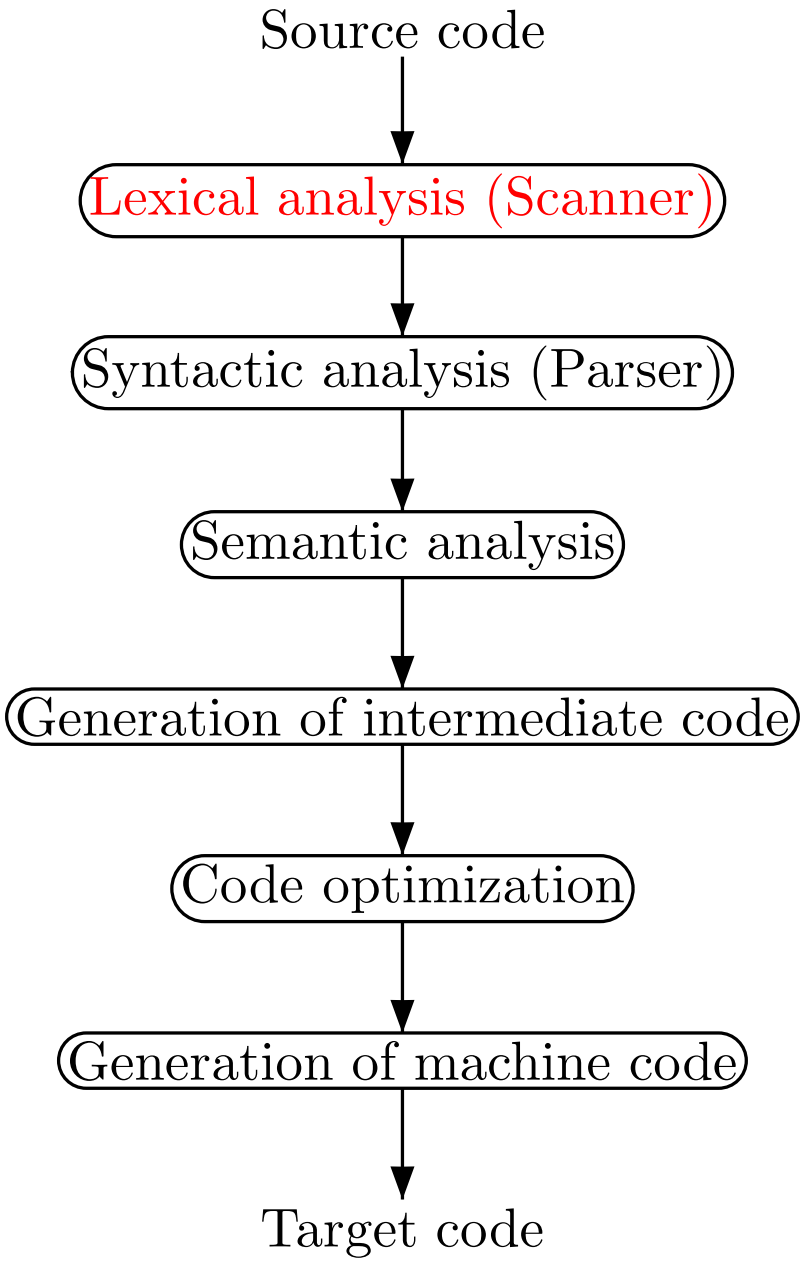
RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc08/

Summer semester 2008

Conceptual Structure of a Compiler



Outline

Lexical Structures

- **Starting point:** source program P as a character sequence
  - Ω (finite) character set (e.g., ASCII, ISO Latin-1, Unicode, ...)
  - $a, b, c, \ldots \in \Omega$ characters (= lexical atoms)
  - $P \in \Omega^*$ source program
    (of course, not every $w \in \Omega^*$ is a valid program)

Lexical Structures

- **Starting point:** source program P as a character sequence
  - Ω (finite) character set (e.g., ASCII, ISO Latin-1, Unicode, ...)
  - $a, b, c, \ldots \in \Omega$ characters (= lexical atoms)
  - $P \in \Omega^*$ source program
    (of course, not every $w \in \Omega^*$ is a valid program)
- P exhibits lexical structures:
  - natural language for keywords, identifiers, ...
  - mathematical notation for numbers, formulae, ...
    (e.g., $x^2 \rightsquigarrow$ `x**2`)
  - spaces, linebreaks, indentation
  - comments and compiler directives (pragmas)
- Translation of P follows its hierarchical structure (later)
- Pragmatic aspects mostly irrelevant (e.g., `x**2` or `x^2` for x^2)

Observations

1. Syntactic atoms (called symbols) are represented as sequences of lexical atoms, called lexemes

First goal of lexical analysis

Decomposition of P into a sequence of lexemes

Observations

1. Syntactic atoms (called symbols) are represented as sequences of lexical atoms, called lexemes

First goal of lexical analysis

Decomposition of P into a sequence of lexemes

2. Differences between similar lexemes are (mostly) irrelevant (e.g., identifiers do not need to be distinguished)
   - lexemes grouped into symbol classes (e.g., identifiers, numbers, ...)
   - symbol classes abstractly represented by tokens
   - symbols identified by additional attributes (e.g., identifier names, numerical values, ...; required for semantic analysis and code generation)

   $\Longrightarrow$ symbol = (token, attribute)

Second goal of lexical analysis

Transformation of a sequence of lexemes into a sequence of symbols

Lexical Analysis

Definition 2.1

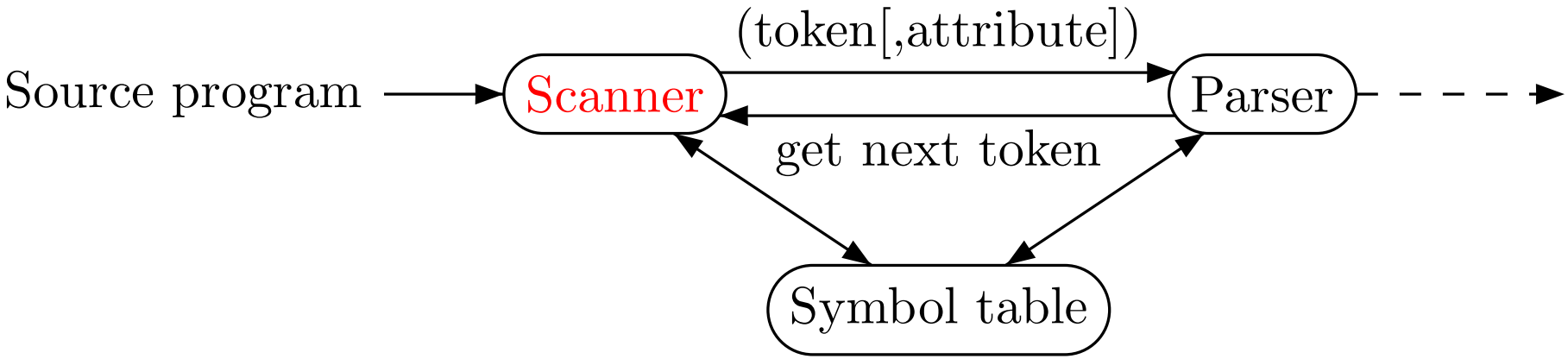
The goal of lexical analysis is to decompose a source program into a sequence of lexemes and their transformation into a sequence of symbols.

Lexical Analysis

Definition 2.1

The goal of lexical analysis is to decompose a source program into a sequence of lexemes and their transformation into a sequence of symbols.

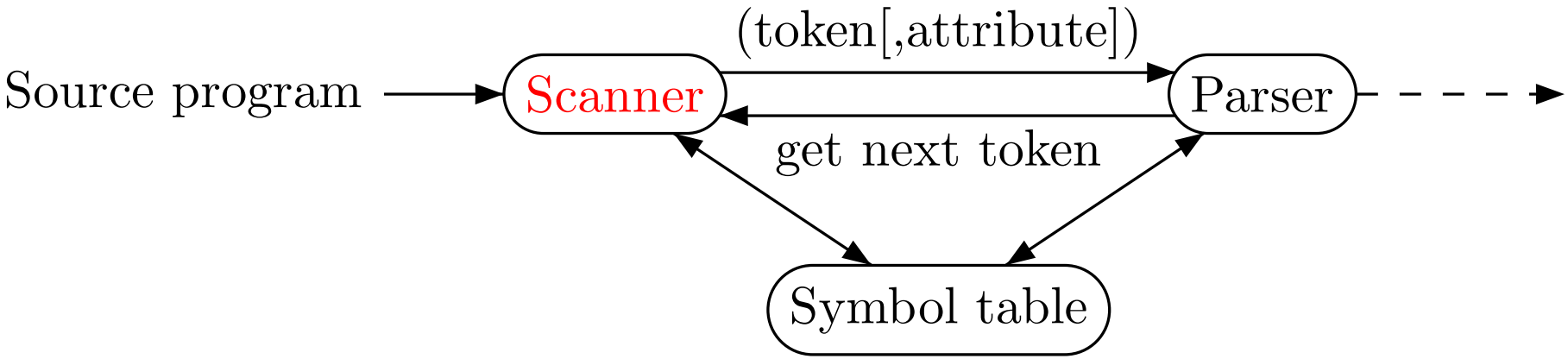
The corresponding program is called a scanner:



Definition 2.1

The goal of lexical analysis is to decompose a source program into a sequence of lexemes and their transformation into a sequence of symbols.

The corresponding program is called a scanner:



Example:

$$\ldots\,\text{␣x1␣:=y2+␣1␣;␣}\ldots$$

$$\Downarrow$$

$$\ldots(\mathsf{id}, p_1)(\mathsf{gets},)(\mathsf{id}, p_2)(\mathsf{plus},)(\mathsf{int}, 1)(\mathsf{sem},)\ldots$$

Important Classes of Symbols

Identifiers:

- for naming variables, constants, types, procedures, classes, ...
- usually a sequence of letters and digits, starting with a letter
- keywords usually forbidden; length possibly restricted

Important Classes of Symbols

Identifiers:

- for naming variables, constants, types, procedures, classes, ...
- usually a sequence of letters and digits, starting with a letter
- keywords usually forbidden; length possibly restricted

Keywords:

- identifiers with a predefined meaning
- for representing control structures (`while`), operators (`and`), ...

Important Classes of Symbols

Identifiers:

- for naming variables, constants, types, procedures, classes, ...
- usually a sequence of letters and digits, starting with a letter
- keywords usually forbidden; length possibly restricted

Keywords:

- identifiers with a predefined meaning
- for representing control structures (`while`), operators (`and`), ...

Numerals: certain sequences of digits, `+`, `-`, letters (for exponent and hexadecimal representation)

Important Classes of Symbols

Identifiers:
- for naming variables, constants, types, procedures, classes, ...
- usually a sequence of letters and digits, starting with a letter
- keywords usually forbidden; length possibly restricted

Keywords:
- identifiers with a predefined meaning
- for representing control structures (`while`), operators (`and`), ...

Numerals: certain sequences of digits, `+`, `-`, letters (for exponent and hexadecimal representation)

Simple symbols:
- one special character, e.g., `+`, `*`, `<`, `(`, `;`, ...
- each makes up a symbol class (plus, ...)

Important Classes of Symbols

Identifiers:
- for naming variables, constants, types, procedures, classes, ...
- usually a sequence of letters and digits, starting with a letter
- keywords usually forbidden; length possibly restricted

Keywords:
- identifiers with a predefined meaning
- for representing control structures (`while`), operators (`and`), ...

Numerals: certain sequences of digits, `+`, `-`, letters (for exponent and hexadecimal representation)

Simple symbols:
- one special character, e.g., `+`, `*`, `<`, `(`, `;`, ...
- each makes up a symbol class (plus, ...)

Composite symbols:
- two or more special characters, e.g., `:=`, `**`, `<=`, ...
- each makes up a symbol class (gets, ...)

Important Classes of Symbols

Identifiers:
- for naming variables, constants, types, procedures, classes, ...
- usually a sequence of letters and digits, starting with a letter
- keywords usually forbidden; length possibly restricted

Keywords:
- identifiers with a predefined meaning
- for representing control structures (`while`), operators (`and`), ...

Numerals: certain sequences of digits, `+`, `-`, letters (for exponent and hexadecimal representation)

Simple symbols:
- one special character, e.g., `+`, `*`, `<`, `(`, `;`, ...
- each makes up a symbol class (plus, ...)

Composite symbols:
- two or more special characters, e.g., `:=`, `**`, `<=`, ...
- each makes up a symbol class (gets, ...)

White spaces:
- blanks, tabs, linebreaks, ...
- usually for separating symbols (exception: FORTRAN)

Specification and Implementation of Scanners

Representation of symbols: symbol = (token, attribute)

Token: (binary) denotation of symbol class (id, gets, plus, ...)

Attribute: additional information required in later compilation phases

- reference to symbol table
- value of numeral
- ...
- usually empty for singleton symbol classes

Specification and Implementation of Scanners

Representation of symbols: symbol = (token, attribute)

Token: (binary) denotation of symbol class (id, gets, plus, ...)

Attribute: additional information required in later compilation phases

- reference to symbol table
- value of numeral
- ...
- usually empty for singleton symbol classes

Observation: symbol classes are regular sets

$\Longrightarrow$

- specification by regular expressions
- recognition by finite automata
- enables automatic generation of scanners (`[f]lex`)

Outline

Regular Expressions I

Definition 2.2 (Syntax of regular expressions)

Given some alphabet Ω, the set of regular expressions over Ω, RE_Ω, is the least set with

- $\Lambda \in RE_\Omega$,
- $\Omega \subseteq RE_\Omega$, and
- whenever $\alpha, \beta \in RE_\Omega$, also $\alpha + \beta, \alpha \cdot \beta, \alpha^* \in RE_\Omega$.

Regular Expressions I

Definition 2.2 (Syntax of regular expressions)

Given some alphabet Ω, the set of regular expressions over Ω, RE_Ω, is the least set with

- $\Lambda \in RE_\Omega$,
- $\Omega \subseteq RE_\Omega$, and
- whenever $\alpha, \beta \in RE_\Omega$, also $\alpha + \beta, \alpha \cdot \beta, \alpha^* \in RE_\Omega$.

Remarks:

- abbreviation: $\alpha^+ := \alpha \cdot \alpha^*$
- $\alpha \cdot \beta$ often written as $\alpha\beta$
- * binds stronger than $\cdot$, $\cdot$ binds stronger than $+$ (i.e., $a + b \cdot c^* := a + (b \cdot (c^*))$)

Regular Expressions II

Regular expressions specify regular languages:

Definition 2.3 (Semantics of regular expressions)

The semantics of a regular expression is defined by the mapping

$$[\![.]\!] : RE_\Omega \to 2^{\Omega^*} \text{ where}$$

$$\begin{aligned} [\![\Lambda]\!] &:= \emptyset \\ [\![a]\!] &:= \{a\} \\ [\![\alpha + \beta]\!] &:= [\![\alpha]\!] \cup [\![\beta]\!] \\ [\![\alpha \cdot \beta]\!] &:= [\![\alpha]\!] \cdot [\![\beta]\!] \\ [\![\alpha^*]\!] &:= [\![\alpha]\!]^* \end{aligned}$$

Regular Expressions II

Regular expressions specify regular languages:

Definition 2.3 (Semantics of regular expressions)

The semantics of a regular expression is defined by the mapping

$$[\![.]\!] : RE_\Omega \to 2^{\Omega^*} \text{ where}$$

$$\begin{aligned} [\![\Lambda]\!] &:= \emptyset \\ [\![a]\!] &:= \{a\} \\ [\![\alpha + \beta]\!] &:= [\![\alpha]\!] \cup [\![\beta]\!] \\ [\![\alpha \cdot \beta]\!] &:= [\![\alpha]\!] \cdot [\![\beta]\!] \\ [\![\alpha^*]\!] &:= [\![\alpha]\!]^* \end{aligned}$$

Remarks: for formal languages $L, M \subseteq \Omega^*$, we have

- $L \cdot M := \{vw \mid v \in L, w \in M\}$
- $L^* := \bigcup_{n=0}^{\infty} L^n$ where $L^0 := \{\varepsilon\}$ and $L^{n+1} := L \cdot L^n$
  $(\implies L^* = \{w_1 w_2 \dots w_n \mid n \in \mathbb{N}, w_i \in L\}$ and $\varepsilon \in L^*)$
- $[\![\Lambda^*]\!] = [\![\Lambda]\!]^* = \emptyset^* = \{\varepsilon\}$

Outline

The Simple Matching Problem I

Problem 2.4 (Simple matching problem)

Given $\alpha \in RE_\Omega$ and $w \in \Omega^*$, decide whether $w \in [\![\alpha]\!]$ or not.

The Simple Matching Problem I

Problem 2.4 (Simple matching problem)

Given $\alpha \in RE_\Omega$ and $w \in \Omega^*$, decide whether $w \in [\![\alpha]\!]$ or not.

This problem can be solved using the following concept:

Definition 2.5 (Finite automaton)

A nondeterministic finite automaton (NFA) is of the form $\mathfrak{A} = \langle Q, \Omega, \delta, q_0, F \rangle$ where

- Q is a finite set of states
- Ω denotes the input alphabet
- $\delta : Q \times \Omega_\varepsilon \to 2^Q$ is the transition function where $\Omega_\varepsilon := \Omega \cup \{\varepsilon\}$
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of final states

The set of all NFA over Ω is denoted by NFA_Ω.
If $\delta(q, \varepsilon) = \emptyset$ and $|\delta(q, a)| = 1$ for every $q \in Q$ and $a \in \Omega$ (i.e., $\delta : Q \times \Omega \to Q$), then $\mathfrak{A}$ is called deterministic (DFA). Notation: DFA_Ω

The Simple Matching Problem II

Definition 2.6 (Acceptance condition)

Let $\mathfrak{A} = \langle Q, \Omega, \delta, q_0, F \rangle \in NFA_\Omega$.

- The ε-closure $\varepsilon(T) \subseteq Q$ of a subset $T \subseteq Q$ is defined by
  - $T \subseteq \varepsilon(T)$ and
  - if $q \in \varepsilon(T)$, then $\delta(q, \varepsilon) \subseteq \varepsilon(T)$

The Simple Matching Problem II

Definition 2.6 (Acceptance condition)

Let $\mathfrak{A} = \langle Q, \Omega, \delta, q_0, F \rangle \in NFA_\Omega$.

- The ε-closure $\varepsilon(T) \subseteq Q$ of a subset $T \subseteq Q$ is defined by
  - $T \subseteq \varepsilon(T)$ and
  - if $q \in \varepsilon(T)$, then $\delta(q, \varepsilon) \subseteq \varepsilon(T)$
- The extended transition function of $\mathfrak{A}$, $\hat{\delta} : 2^Q \times \Omega^* \to 2^Q$, is given by
  - $\hat{\delta}(T, \varepsilon) := \varepsilon(T)$ and
  - $\hat{\delta}(T, wa) := \varepsilon\left(\bigcup_{q \in \hat{\delta}(T,w)} \delta(q, a)\right)$ $(w \in \Omega^*, a \in \Omega)$

The Simple Matching Problem II

Definition 2.6 (Acceptance condition)

Let $\mathfrak{A} = \langle Q, \Omega, \delta, q_0, F \rangle \in NFA_\Omega$.

- The ε-closure $\varepsilon(T) \subseteq Q$ of a subset $T \subseteq Q$ is defined by
  - $T \subseteq \varepsilon(T)$ and
  - if $q \in \varepsilon(T)$, then $\delta(q, \varepsilon) \subseteq \varepsilon(T)$
- The extended transition function of $\mathfrak{A}$, $\hat{\delta} : 2^Q \times \Omega^* \to 2^Q$, is given by
  - $\hat{\delta}(T, \varepsilon) := \varepsilon(T)$ and
  - $\hat{\delta}(T, wa) := \varepsilon\left(\bigcup_{q \in \hat{\delta}(T,w)} \delta(q, a)\right)$ $(w \in \Omega^*, a \in \Omega)$
- $\mathfrak{A}$ recognizes the language

$$L(\mathfrak{A}) := \{w \in \Omega^* \mid \hat{\delta}(\{q_0\}, w) \cap F \neq \emptyset\}$$

The Simple Matching Problem II

Definition 2.6 (Acceptance condition)

Let $\mathfrak{A} = \langle Q, \Omega, \delta, q_0, F \rangle \in NFA_\Omega$.

- The ε-closure $\varepsilon(T) \subseteq Q$ of a subset $T \subseteq Q$ is defined by
  - $T \subseteq \varepsilon(T)$ and
  - if $q \in \varepsilon(T)$, then $\delta(q, \varepsilon) \subseteq \varepsilon(T)$
- The extended transition function of $\mathfrak{A}$, $\hat{\delta} : 2^Q \times \Omega^* \to 2^Q$, is given by
  - $\hat{\delta}(T, \varepsilon) := \varepsilon(T)$ and
  - $\hat{\delta}(T, wa) := \varepsilon\left(\bigcup_{q \in \hat{\delta}(T,w)} \delta(q, a)\right)$ $(w \in \Omega^*, a \in \Omega)$
- $\mathfrak{A}$ recognizes the language

$$L(\mathfrak{A}) := \{w \in \Omega^* \mid \hat{\delta}(\{q_0\}, w) \cap F \neq \emptyset\}$$

Example 2.7

NFA for $a^*b + a^*$ (on the board)

The Simple Matching Problem III

Remarks:

- NFA as specified in Definition 2.5 are sometimes called NFA with ε-transitions (ε-NFA).

The Simple Matching Problem III

Remarks:

- NFA as specified in Definition 2.5 are sometimes called NFA with ε-transitions (ε-NFA).
- For $\mathfrak{A} \in DFA_\Omega$, the acceptance condition yields $\hat{\delta} : Q \times \Omega^* \to Q$ with $\hat{\delta}(q, \varepsilon) = q$ and $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$, and

$$L(\mathfrak{A}) = \{w \in \Omega^* \mid \hat{\delta}(q_0, w) \in F\}.$$

The DFA Method

Known from *Automata Theory and Formal Languages*:

Algorithm 2.8 (DFA method)

Input: *regular expression* $\alpha \in RE_\Omega$, *input string* $w \in \Omega^*$

The DFA Method

Known from *Automata Theory and Formal Languages*:

Algorithm 2.8 (DFA method)

Input: *regular expression* $\alpha \in RE_\Omega$, *input string* $w \in \Omega^*$

Procedure:

1. *using Kleene's Theorem, construct* $\mathfrak{A}_\alpha \in NFA_\Omega$ *such that* $L(\mathfrak{A}_\alpha) = [\![\alpha]\!]$
2. *apply powerset construction to obtain* $\mathfrak{A}'_\alpha = \langle Q', \Omega, \delta', q'_0, F' \rangle \in DFA_\Omega$ *with* $L(\mathfrak{A}'_\alpha) = L(\mathfrak{A}_\alpha) = [\![\alpha]\!]$
3. *solve the matching problem by deciding whether* $\hat{\delta}'(q'_0, w) \in F'$

The DFA Method

Known from *Automata Theory and Formal Languages*:

Algorithm 2.8 (DFA method)

Input: *regular expression* $\alpha \in RE_\Omega$, *input string* $w \in \Omega^*$

Procedure:

1. *using Kleene's Theorem, construct* $\mathfrak{A}_\alpha \in NFA_\Omega$ *such that* $L(\mathfrak{A}_\alpha) = [\![\alpha]\!]$
2. *apply powerset construction to obtain* $\mathfrak{A}'_\alpha = \langle Q', \Omega, \delta', q'_0, F' \rangle \in DFA_\Omega$ *with* $L(\mathfrak{A}'_\alpha) = L(\mathfrak{A}_\alpha) = [\![\alpha]\!]$
3. *solve the matching problem by deciding whether* $\hat{\delta}'(q'_0, w) \in F'$

Output: *"yes" or "no"*

The DFA Method

Known from *Automata Theory and Formal Languages*:

Algorithm 2.8 (DFA method)

Input: *regular expression* $\alpha \in RE_\Omega$, *input string* $w \in \Omega^*$

Procedure:

1. *using Kleene's Theorem, construct* $\mathfrak{A}_\alpha \in NFA_\Omega$ *such that* $L(\mathfrak{A}_\alpha) = [\![\alpha]\!]$
2. *apply powerset construction to obtain* $\mathfrak{A}'_\alpha = \langle Q', \Omega, \delta', q'_0, F' \rangle \in DFA_\Omega$ *with* $L(\mathfrak{A}'_\alpha) = L(\mathfrak{A}_\alpha) = [\![\alpha]\!]$
3. *solve the matching problem by deciding whether* $\hat{\delta}'(q'_0, w) \in F'$

Output: *"yes" or "no"*

Example 2.9

1. Kleene's Theorem (on the board)
2. Powerset construction (on the board)

Time and Space Complexity of DFA Method

1. in construction phase:
   - Kleene method: time and space $\mathcal{O}(|\alpha|)$ ($|\alpha|$:= length of α)
   - Powerset construction: time and space $\mathcal{O}(2^{|\mathfrak{A}_\alpha|}) = \mathcal{O}(2^{|\alpha|})$ ($|\mathfrak{A}_\alpha|$:= # of states)

Time and Space Complexity of DFA Method

1. in construction phase:
   - Kleene method: time and space $\mathcal{O}(|\alpha|)$ ($|\alpha| :=$ length of α)
   - Powerset construction: time and space $\mathcal{O}(2^{|\mathfrak{A}_\alpha|}) = \mathcal{O}(2^{|\alpha|})$ ($|\mathfrak{A}_\alpha| :=$ # of states)
2. at runtime:
   - Word problem: time $\mathcal{O}(|w|)$ ($|w| :=$ length of w), space $\mathcal{O}(1)$ (but $\mathcal{O}(2^{|\alpha|})$ for storing DFA)

Time and Space Complexity of DFA Method

1. in construction phase:
   - Kleene method: time and space $\mathcal{O}(|\alpha|)$ ($|\alpha| :=$ length of α)
   - Powerset construction: time and space $\mathcal{O}(2^{|\mathfrak{A}_\alpha|}) = \mathcal{O}(2^{|\alpha|})$ ($|\mathfrak{A}_\alpha| :=$ # of states)
2. at runtime:
   - Word problem: time $\mathcal{O}(|w|)$ ($|w| :=$ length of w), space $\mathcal{O}(1)$ (but $\mathcal{O}(2^{|\alpha|})$ for storing DFA)

$\Longrightarrow$ nice runtime behavior but memory requirements too high (and exponential time in construction phase)

The NFA Method

Idea: decrease memory requirements by applying powerset construction at runtime, i.e., only "to the run of w through $\mathfrak{A}_\alpha$" (direct computation of $\hat{\delta}(\{q_0\}, w)$; see Example 2.7)

The NFA Method

Idea: decrease memory requirements by applying powerset construction at runtime, i.e., only "to the run of w through $\mathfrak{A}_\alpha$" (direct computation of $\hat{\delta}(\{q_0\}, w)$; see Example 2.7)

Algorithm 2.10 (NFA method)

Input: *automaton* $\mathfrak{A}_\alpha = \langle Q, \Omega, \delta, q_0, F \rangle \in NFA_\Omega$,
input string $w \in \Omega^*$

The NFA Method

Idea: decrease memory requirements by applying powerset construction at runtime, i.e., only "to the run of w through $\mathfrak{A}_\alpha$" (direct computation of $\hat{\delta}(\{q_0\}, w)$; see Example 2.7)

Algorithm 2.10 (NFA method)

Input: *automaton* $\mathfrak{A}_\alpha = \langle Q, \Omega, \delta, q_0, F \rangle \in NFA_\Omega$,
input string $w \in \Omega^*$

Variables: $T \subseteq Q$, $a \in \Omega$, $w' \in \Omega^*$

Procedure:
$T := \varepsilon(\{q_0\})$;
while $w \neq \varepsilon$ **do**
  $aw' := w$;
  $T := \varepsilon\left(\bigcup_{q \in T} \delta(q, a)\right)$;
  $w := w'$
od

The NFA Method

Idea: decrease memory requirements by applying powerset construction at runtime, i.e., only "to the run of w through $\mathfrak{A}_\alpha$" (direct computation of $\hat{\delta}(\{q_0\}, w)$; see Example 2.7)

Algorithm 2.10 (NFA method)

Input: *automaton* $\mathfrak{A}_\alpha = \langle Q, \Omega, \delta, q_0, F \rangle \in NFA_\Omega$, *input string* $w \in \Omega^*$

Variables: $T \subseteq Q$, $a \in \Omega$, $w' \in \Omega^*$

Procedure:
$T := \varepsilon(\{q_0\})$;
while $w \neq \varepsilon$ **do**
  $aw' := w$;
  $T := \varepsilon\left(\bigcup_{q \in T} \delta(q, a)\right)$;
  $w := w'$
od

Output: *if* $T \cap F \neq \emptyset$ *then "yes" else "no"*

Complexity Analysis

For NFA Method at runtime:

- Space: $\mathcal{O}(|\alpha|)$ (for storing NFA and T)
- Time: $\mathcal{O}(|\alpha| \cdot |w|)$
  (in the loop's body, $|T|$ states need to be considered)

$\Longrightarrow$ trades exponential space for increase in time

Complexity Analysis

For NFA Method at runtime:

- Space: $\mathcal{O}(|\alpha|)$ (for storing NFA and T)
- Time: $\mathcal{O}(|\alpha| \cdot |w|)$
  (in the loop's body, $|T|$ states need to be considered)

$\Longrightarrow$ trades exponential space for increase in time

Comparison:

Method	Space	Time (for "$w \in [\![\alpha]\!]$?")
DFA	$\mathcal{O}(2^{\|\alpha\|})$	$\mathcal{O}(\|w\|)$
NFA	$\mathcal{O}(\|\alpha\|)$	$\mathcal{O}(\|\alpha\| \cdot \|w\|)$

Complexity Analysis

For NFA Method at runtime:

- Space: $\mathcal{O}(|\alpha|)$ (for storing NFA and T)
- Time: $\mathcal{O}(|\alpha| \cdot |w|)$
  (in the loop's body, $|T|$ states need to be considered)

$\Longrightarrow$ trades exponential space for increase in time

Comparison:

Method	Space	Time (for "$w \in [\![\alpha]\!]$?")
DFA	$\mathcal{O}(2^{\|\alpha\|})$	$\mathcal{O}(\|w\|)$
NFA	$\mathcal{O}(\|\alpha\|)$	$\mathcal{O}(\|\alpha\| \cdot \|w\|)$

In practice:

- Exponential blowup of DFA methode usually does not occur in "real" applications ($\Longrightarrow$ used in `[f]lex`)
- Improvement of NFA method: caching of transitions $\hat{\delta}(T, a)$
  $\Longrightarrow$ combination of both methods