



Compiler Construction

Lecture 2: Lexical Analysis I (Simple Matching Problem)

Summer Semester 2017

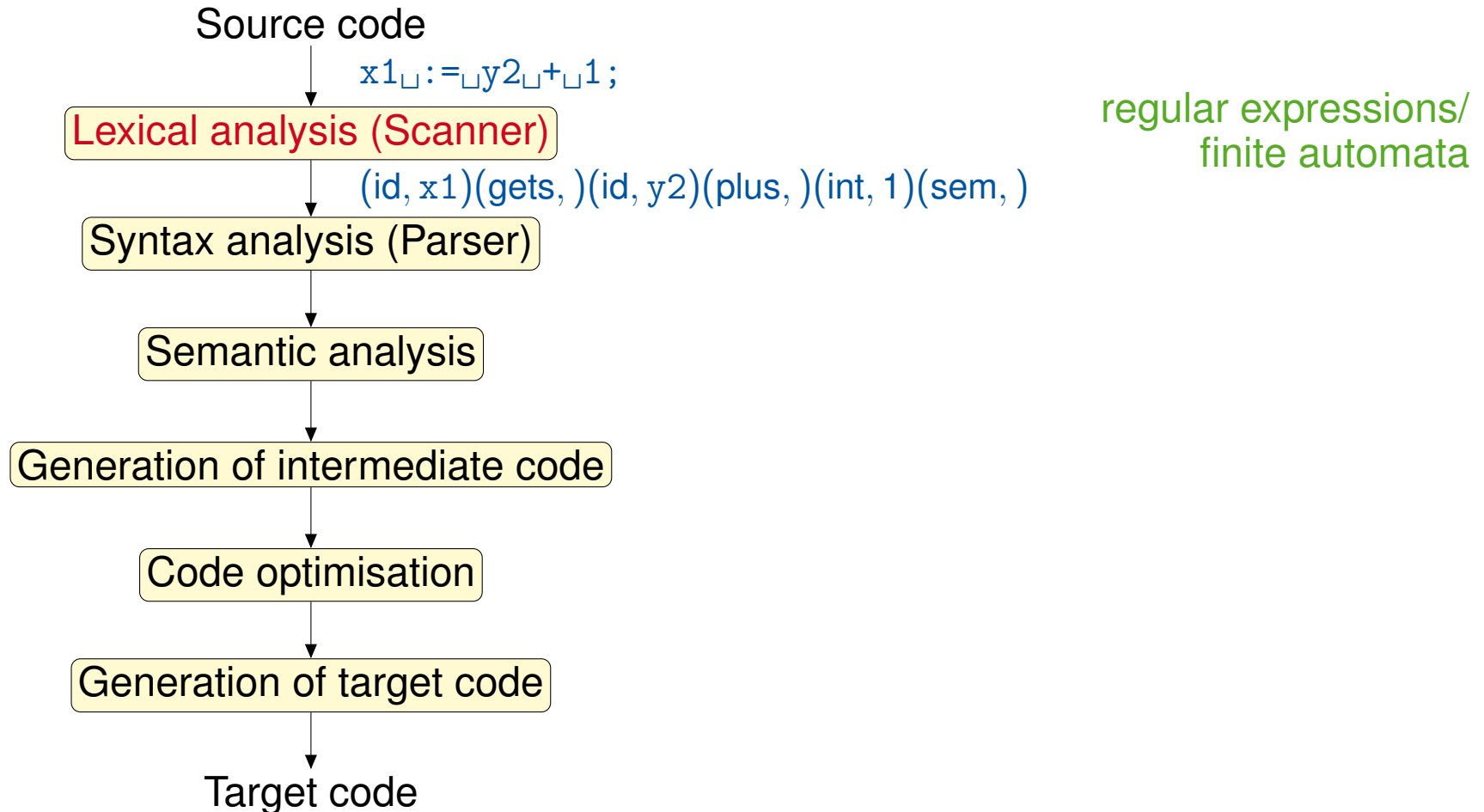
Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-17/cc/>

Conceptual Structure of a Compiler



Problem Statement

Lexical Structures

From Merriam-Webster's Online Dictionary

Lexical: of or relating to words or the vocabulary of a language as distinguished from its grammar and construction

- **Starting point:** source program P as a **character sequence**
 - Ω (finite) **character set** (e.g., ASCII, ISO Latin-1, Unicode, ...)
 - $a, b, c, \dots \in \Omega$ **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, ...
 - textual notation for numbers, formulae, ... (e.g., $x^2 \rightsquigarrow x**2$ or $2.9979 \cdot 10^8 \rightsquigarrow 2.9979D+8$)
 - spaces, line breaks, indentation
 - comments and compiler directives (pragmas)
- Translation of P follows its **hierarchical structure** (later)

Problem Statement

Lexical as Part of Syntax Analysis

Remark: lexical analysis could be made **integral part of syntax analysis** (as regular languages are a proper subclass of context-free languages – cf. ANTLR approach)

Reasons for keeping lexical and syntax analysis separate

Efficiency: scanner may do simple parts of the work faster than a more general parser

Modularity: syntax definition not cluttered with low-level details such as white spaces or comments

Tradition: language standards typically separate lexical and syntactical elements

Problem Statement

Observations

1. Syntactic atoms (called **symbols**) are represented as sequences of input characters, called **lexemes**

First goal of lexical analysis

Decomposition of program text into a **sequence of lexemes**

2. Differences between similar lexemes are (mostly) irrelevant for syntax analysis (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

⇒ **symbol = (token, attribute)**

Second goal of lexical analysis

Transformation of a sequence of lexemes into a **sequence of symbols**

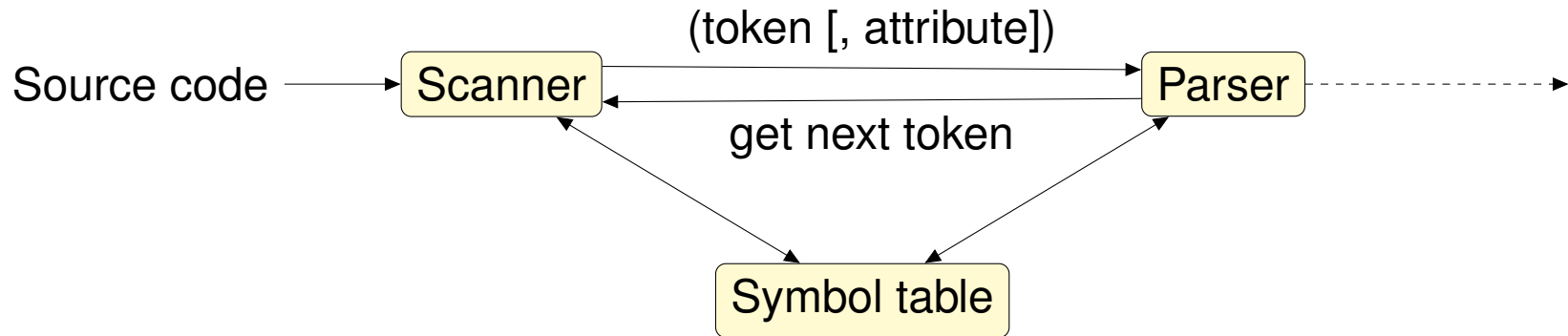
Problem Statement

Lexical Analysis

Definition 2.1

The goal of **lexical analysis** is the decomposition a source program into a sequence of lexemes and their transformation into a sequence of symbols.

The corresponding program is called a **scanner** (or **lexer**):



Example:

... `x1 := y2 + 1;` ...
↓
... (id, p_1)(gets,)(id, p_2)(plus,)(int, 1)(sem,) ...

Problem Statement

Important Symbol Classes

- Identifiers:**
- for naming variables, constants, types, procedures, classes, ...
 - usually a sequence of letters and digits (and possibly special symbols), 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)

- Special symbols:**
- one special character, e.g., `+`, `*`, `<`, `(`, `;`, ...
 - ... or two or more special characters, e.g., `:=`, `**`, `<=`, ...
 - each makes up a symbol class (`plus`, `gets`, ...)
 - ... or several combined into one class (`arithOp`)

- White spaces:**
- blanks, tabs, line breaks, ...
 - generally for separating symbols (exception: FORTRAN)
 - usually not represented by token (but just removed)

Problem Statement

Specification and Implementation of Scanners

Representation of symbols: $\text{symbol} = (\text{token}, \text{attribute})$

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

Attribute: additional information required in later compilation phases

- reference to symbol table,
- value of numeral,
- concrete arithmetic/relational/Boolean operator, ...
- usually unused for singleton symbol classes

Observation: symbol classes are **regular sets**

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

Specification of Symbol Classes

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

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

Remarks:

- abbreviations: $\alpha^+ := \alpha \cdot \alpha^*$, $\varepsilon := \emptyset^*$
- $\alpha \cdot \beta$ often written as $\alpha\beta$
- Binding priority: $* > \cdot > \mid$ (i.e., $a \mid b \cdot c^* := a \mid (b \cdot (c^*))$)

Specification of Symbol Classes

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 $\llbracket \cdot \rrbracket : RE_{\Omega} \rightarrow 2^{\Omega^*}$:

$$\begin{aligned}\llbracket \emptyset \rrbracket &:= \emptyset \\ \llbracket a \rrbracket &:= \{a\} \\ \llbracket \alpha \mid \beta \rrbracket &:= \llbracket \alpha \rrbracket \cup \llbracket \beta \rrbracket \\ \llbracket \alpha \cdot \beta \rrbracket &:= \llbracket \alpha \rrbracket \cdot \llbracket \beta \rrbracket \\ \llbracket \alpha^* \rrbracket &:= \llbracket \alpha \rrbracket^*\end{aligned}$$

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 $L^{n+1} := L \cdot L^n$
 - thus $L^* = \{w_1 w_2 \dots w_n \mid n \in \mathbb{N}, \forall 1 \leq i \leq n : w_i \in L\}$ and $\varepsilon \in L^*$
- $\llbracket \emptyset^* \rrbracket = \llbracket \emptyset \rrbracket^* = \emptyset^* = \{\varepsilon\}$

Regular Expressions III

Example 2.4

1. A keyword: `begin`

2. Identifiers:

$$(a \mid \dots \mid z \mid A \mid \dots \mid Z)(a \mid \dots \mid z \mid A \mid \dots \mid Z \mid 0 \mid \dots \mid 9 \mid \$ \mid _ \mid \dots)^*$$

3. (Unsigned) Integer numbers: $(0 \mid \dots \mid 9)^+$

4. (Unsigned) Fixed-point numbers:

$$((0 \mid \dots \mid 9)^+ \cdot (0 \mid \dots \mid 9)^*) \mid ((0 \mid \dots \mid 9)^* \cdot (0 \mid \dots \mid 9)^+)$$

The Simple Matching Problem

The Simple Matching Problem I

Problem 2.5 (Simple matching problem)

Given $\alpha \in RE_{\Omega}$ and $w \in \Omega^*$, decide whether $w \in \llbracket \alpha \rrbracket$ or not.

This problem can be solved using the following concept:

Definition 2.6 (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**
- $\delta : Q \times \Omega_{\varepsilon} \rightarrow 2^Q$ is the **transition function** with $\Omega_{\varepsilon} := \Omega \cup \{\varepsilon\}$ (write $q \xrightarrow{x} q'$ for $q' \in \delta(q, x)$)
- $q_0 \in Q$ 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., $\delta : Q \times \Omega \rightarrow Q$), then \mathfrak{A} is called **deterministic (DFA)**. Notation: DFA_{Ω}

The Simple Matching Problem

The Simple Matching Problem II

Definition 2.7 (Acceptance condition)

Let $\mathcal{A} = \langle Q, \Omega, \delta, q_0, F \rangle \in \text{NFA}_\Omega$ and $w = a_1 \dots a_n \in \Omega^*$.

- A w -labelled \mathcal{A} -run from q_1 to q_2 is a sequence of transitions

$$q_1 \xrightarrow{\varepsilon^*} \xrightarrow{a_1} \xrightarrow{\varepsilon^*} \xrightarrow{a_2} \xrightarrow{\varepsilon^*} \dots \xrightarrow{\varepsilon^*} \xrightarrow{a_n} \xrightarrow{\varepsilon^*} q_2$$

- \mathcal{A} **accepts** w if there is a w -labelled \mathcal{A} -run from q_0 to some $q \in F$
- The **language** recognised by \mathcal{A} is

$$L(\mathcal{A}) := \{w \in \Omega^* \mid \mathcal{A} \text{ accepts } w\}$$

- A language $L \subseteq \Omega^*$ is called **NFA-recognisable** if there exists a NFA \mathcal{A} such that $L(\mathcal{A}) = L$

Example 2.8

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

The Simple Matching Problem

The Simple Matching Problem III

Remarks:

- NFA as specified in Definition 2.6 are sometimes called **NFA with ε -transitions (ε -NFA)**.
- For $\mathfrak{A} \in DFA_{\Omega}$, the acceptance condition yields $\delta^* : Q \times \Omega^* \rightarrow Q$ with $\delta^*(q, \varepsilon) = q$ and $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$, and

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

The Simple Matching Problem

The DFA Method I

Known from *Formal Systems, Automata and Processes*:

Algorithm 2.9 (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 $L(\mathfrak{A}_{\alpha}) = \llbracket \alpha \rrbracket$

2. apply **powerset construction** (cf. Definition 2.11) to obtain

$\mathfrak{A}'_{\alpha} = \langle Q', \Omega, \delta', q'_0, F' \rangle \in DFA_{\Omega}$ with $L(\mathfrak{A}'_{\alpha}) = L(\mathfrak{A}_{\alpha}) = \llbracket \alpha \rrbracket$

3. solve the **matching problem** by deciding whether $\delta'^*(q'_0, w) \in F'$

Output: “yes” or “no”

Kleene's Theorem

Working principle: on the board

The Simple Matching Problem

The DFA Method II

The powerset construction involves the following concept:

Definition 2.10 (ε -closure)

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 the least set with (1) $T \subseteq \varepsilon(T)$ and (2) if $q \in \varepsilon(T)$, then $\delta(q, \varepsilon) \subseteq \varepsilon(T)$

Definition 2.11 (Powerset construction)

Let $\mathfrak{A} = \langle Q, \Omega, \delta, q_0, F \rangle \in NFA_{\Omega}$. The **powerset automaton** $\mathfrak{A}' = \langle Q', \Omega, \delta', q'_0, F' \rangle \in DFA_{\Omega}$ is defined by

- $Q' := 2^Q$
- $q'_0 := \varepsilon(\{q_0\})$
- $\forall T \subseteq Q, a \in \Omega : \delta'(T, a) := \varepsilon(\bigcup_{q \in T} \delta(q, a))$
- $F' := \{T \subseteq Q \mid T \cap F \neq \emptyset\}$

Example 2.12

Powerset construction for Example 2.8 (on the board)

Complexity Analysis of Simple Matching

Complexity of DFA Method

1. in construction phase:

- **Kleene method**: time and space $\mathcal{O}(|\alpha|)$
(where $|\alpha| := \text{length of } \alpha$)
- **Powerset construction**: time and space $\mathcal{O}(2^{|\mathcal{A}_\alpha|}) = \mathcal{O}(2^{|\alpha|})$
(where $|\mathcal{A}_\alpha| := \# \text{ of states of } \mathcal{A}_\alpha$)

2. at runtime:

- **Word problem**:
 - time $\mathcal{O}(|w|)$ (where $|w| := \text{length of } w$)
 - space $\mathcal{O}(1)$ (but $\mathcal{O}(2^{|\alpha|})$ for storing DFA)

⇒ nice runtime behaviour but memory requirements very high
(and exponential time in construction phase)

Complexity Analysis of Simple Matching

The NFA Method

Idea: reduce memory requirements by **applying powerset construction at runtime**, i.e., only “to the run of w through \mathcal{A}_α ”

Algorithm 2.13 (NFA method)

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

Variables: $T \subseteq Q$, $a \in \Omega$

Procedure: $T := \varepsilon(\{q_0\});$

while $w \neq \varepsilon$ **do**

$a := \text{head}(w);$

$T := \varepsilon \left(\bigcup_{q \in T} \delta(q, a) \right);$

$w := \text{tail}(w)$

od

Output: if $T \cap F \neq \emptyset$ then “yes” else “no”

Complexity Analysis of Simple Matching

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)

Comparison:

Method	Space	Time (for “ $w \in \llbracket \alpha \rrbracket$?”)
DFA	$\mathcal{O}(2^{ \alpha })$	$\mathcal{O}(w)$
NFA	$\mathcal{O}(\alpha)$	$\mathcal{O}(\alpha \cdot w)$

\implies trades exponential space for increase in time

In practice:

- Exponential blowup of DFA method does usually not occur in practice
(\implies used in `[f]lex`)
- Improvement of NFA method: caching of transitions $\delta'(T, a)$
 \implies combination of both methods