Theoretical Computer Science

Formal Languages

Technische Hochschule Rosenheim
Sommer 2022
Prof. Dr. Jochen Schmidt

Overview

- Definition of formal languages & grammars
- Chomsky hierarchy
- Regular expressions
- Pumping lemma

Definition of Formal Languages

Introduction

Technische Hochschule **Rosenheim**

- Natural languages are not suitable for programming computers
- Therefore: Development of special programming languages
- Most noticeable difference to natural languages:
  - strictly formalized rules of programming languages
  - small language regarding
    - Vocabulary and
    - Rules
- Here: Basic properties of formal languages (*formale Sprachen*)
- These are the theoretical foundations of programming languages and compilers

- Formal Language: A subset $L \subseteq \Sigma^*$ over a finite alphabet Σ
  - we have seen this in the previous chapters: recognized language of an automaton
  - L contains the words of the language – it does not say anything about how to built them
  - for generating words we need a formal grammar

Example

- Language $L = \{10^n1 \mid n \in \mathbb{N}_0\}$

- Grammar for generating this language
  Terminal symbols: $\{0, 1\}$ — customary: use lowercase letters, numbers
  Nonterminal symbols: $\{S, A\}$ — customary: use capital letters
  Start symbol: S
  Production rules (replacement of strings):

  $S \longrightarrow 1A1,\quad S \longrightarrow 11$
  $A \longrightarrow 0A,\quad A \longrightarrow 0$

  customary: use BNF (Backus Naur form, | = "or")
  $S \longrightarrow 11 \mid 1A1$
  $A \longrightarrow 0A \mid 0$

- Derivation of 100001:
  $S \Rightarrow 1A1 \Rightarrow 10A1 \Rightarrow 100A1 \Rightarrow 1000A1 \Rightarrow 100001$
  "$\Rightarrow$": single step, apply one rule

  $S \Rightarrow^* 100001$
  "$\Rightarrow^*$": multiple steps

Definition: Formal Grammar/Language

Formal Grammar (*formale Grammatik*)

- Finite alphabet Σ of terminal symbols (*Terminalsymbole*)
- Finite alphabet V of nonterminal symbols (*Nichtterminalsymbole*) or syntactic variables
  - V contains at least the start symbol S
- $V \cap \Sigma = \emptyset$

- Finite set P of production rules ("productions", *Produktionen*), i.e., derivation rules
  u $\longrightarrow$ v where u $\in (V \cup \Sigma)^* V (V \cup \Sigma)^*$, v $\in (V \cup \Sigma)^*$
  ("replace u by v")
  This is the syntax defined by the grammar.

Formal language (*formale Sprache*) $L(G) = \{w \mid w \in \Sigma^*, S \Rightarrow^* w\}$
All words w consisting only of terminal symbols that can be derived from the start symbol S by using the production rules of grammar G

Formal Languages, Grammars, and Automata

Grammar

top-down

generates words of

Language

accepts words of

bottom-up

Automaton

Grammar	Start symbol	Terminal symbols	Nonterminal symbols	Production rules
Automaton	Initial State	Input alphabet	States	State transitions

Chomsky Hierarchy

Overview

Significant contributions to the classification of formal languages by:

- the Norwegian mathematician A. Thue (1863 - 1922)
- and since about 1955 by the American linguist Noam Chomsky (*1928)

- Classification of grammars and languages into the so-called Chomsky hierarchy
  - by type of permitted productions from type 0 (most general) to type 3 (most restricted)

Chomsky-Hierarchy (Grammars)

- Type 0 (unrestricted grammar)
  - no restrictions on production rules: Productions have the form
    xAy $\rightarrow$ u where A$\in V^+$ and x, y, u $\in (V \cup \Sigma)^*$

- Type 1 (context-sensitive grammar)
  - Productions have the form:
    xAy $\rightarrow$ xuy where x, y $\in (V \cup \Sigma)^*$, A $\in V$ and u $\in (V \cup \Sigma)^+$ or
    S $\rightarrow$ ε (S = start symbol; if used, S must not appear on the right side of any rule)
  - Notes:
    - except for S $\rightarrow$ ε these rules are monotonic (noncontracting, the strings can only get longer)
    - in fact, all monotonic grammars define the same language as context-sensitive grammars:
      u $\rightarrow$ v, |u| ≤ |v| u, v $\in (V \cup \Sigma)^*$
    - however, the production rules then no longer necessarily are of the above form
    - some authors (e.g., Schöning) use this as a definition of type 1 grammars

Chomsky-Hierarchy (Grammars)

Technische Hochschule Rosenheim

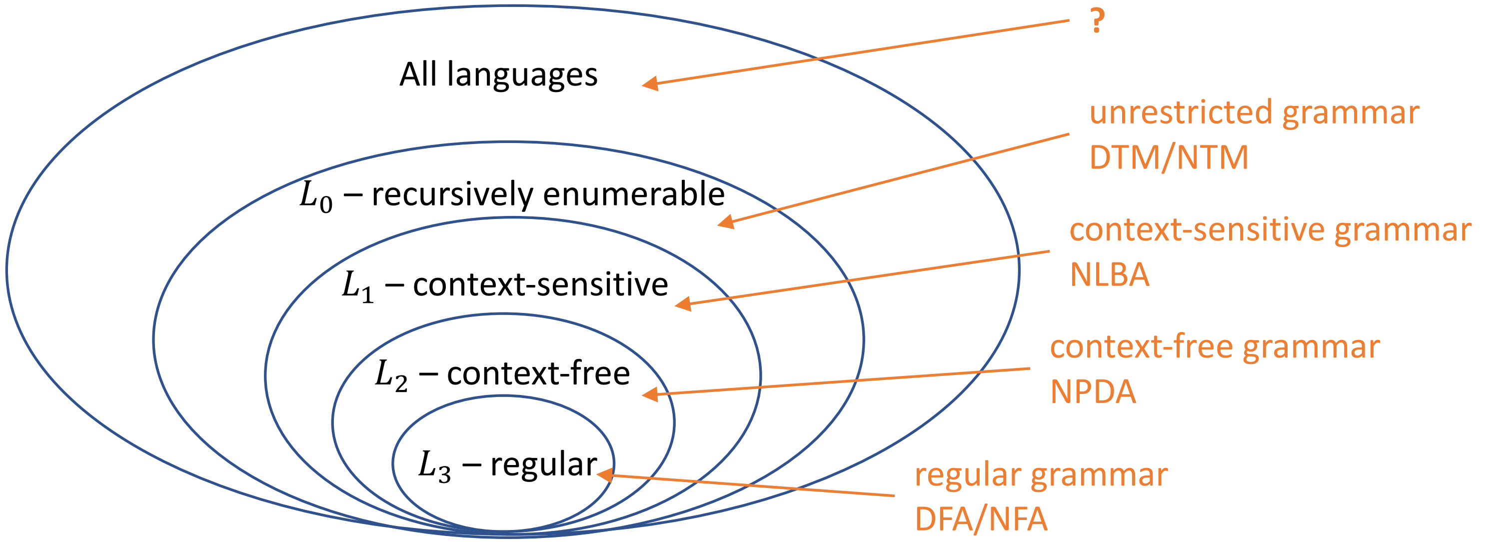
- Type 2 (context-free grammar)
  - Productions have the form:
    A $\longrightarrow$ u where A $\in V$ and u $\in (V \cup \Sigma)^+$ or
    S $\longrightarrow$ ε (S = start symbol; if used, S must not appear on the right side of any rule)

- Type 3 (regular grammar)
  - Productions have the form:
    S $\longrightarrow$ ε (S = start symbol; if used, S must not appear on the right side of any rule) or
    A $\longrightarrow$ u where A $\in V$ and for all rules either u $\in \Sigma^+ \cup \Sigma^+ V$ or u $\in \Sigma^+ \cup V\Sigma^+$, i.e.:
    - right-linear productions:
      A $\longrightarrow$ uB where A,B $\in V$ and u $\in \Sigma^+$
    - left-linear productions :
      A $\longrightarrow$ Bu where A,B $\in V$ and u $\in \Sigma^+$
    - terminal productions :
      A $\longrightarrow$ u where A $\in V$ and u $\in \Sigma^+$
  - Productions must be either **all** right-linear or **all** left-linear!

Chomsky-Hierarchy (Languages)

- A language L is said to be of type i (i = 0, 1, 2, 3), if a Chomsky grammar G of type i exists, with $L(G) = L$

- Note: The keyword is "exists"
  - a language remains being of type i even if you specify a grammar of type j with j < i
  - e.g., you could specify a context-sensitive grammar for a regular language



Example

$L = \{10^n1 \mid n \in \mathbb{N}_0\}$

- Start symbol: S

- Grammar, type 0: $S \longrightarrow 1A1, \quad A \longrightarrow 0A \mid A \longrightarrow \varepsilon$
  Derivation of 100001: $S \Rightarrow 1A1 \Rightarrow 10A1 \Rightarrow 100A1 \Rightarrow 1000A1 \Rightarrow 10000A1 \Rightarrow 100001$

- Grammar, type 2: $S \longrightarrow 11 \mid 1A1, \quad A \longrightarrow 0A \mid A \longrightarrow 0$
  Derivation of 100001: $S \Rightarrow 1A1 \Rightarrow 10A1 \Rightarrow 100A1 \Rightarrow 1000A1 \Rightarrow 100001$

- Grammar, type 3: $S \longrightarrow 11 \mid 1A, \quad A \longrightarrow 0A \mid A \longrightarrow 0B, \quad B \longrightarrow 1$
  Derivation of 100001: $S \Rightarrow 1A \Rightarrow 10A \Rightarrow 100A \Rightarrow 1000A \Rightarrow 10000B \Rightarrow 100001$

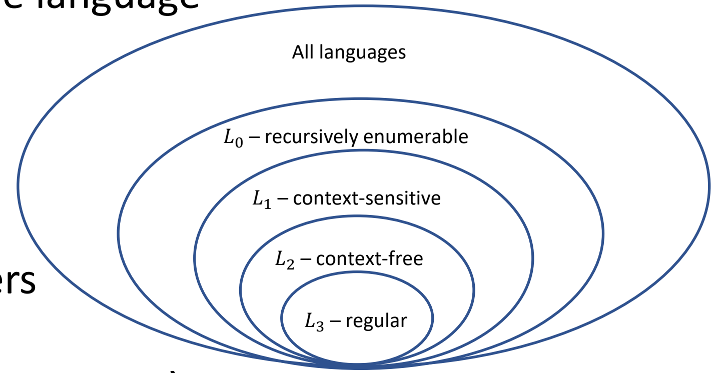
- The language L is regular
  - it is also context-free, context-sensitive, recursively enumerable – each superclass contains the others
  - what we are interested in is the most restrictive language class

Are There Really Languages More General Than Type 0?

Technische Hochschule Rosenheim

There must be!

- Type 0 languages are called recursively enumerable for a reason:
  - their words can be systematically generated one after the other by an algorithm (e.g., the grammar)
  - so there exists a mapping of the natural numbers to the set of words of an enumerable language (we can count the words)
- By definition, each subset $L \subseteq \Sigma^*$ is a language
- All languages = the set of all subsets of Σ^* (= the powerset, *Potenzmenge*)
  - Σ^* is a countable set (we can map it to the natural numbers, it is enumerable)
  - the powerset of a countable set is uncountable = larger than the set of natural numbers
- So: Type 0 languages = countable $\longleftrightarrow$ set of all languages = uncountable: There are more languages than those that are of type 0 (in fact: infinitely many more)



- Conclusion:
  - There are languages that cannot be generated by a grammar
  - and that cannot be recognized by a Turing Machine (and therefore a computer).

There must exist problems that we cannot (and will **never** be able to) solve with a computer!

Summary Equivalency: Grammar – Automaton

Technische Hochschule Rosenheim

Language	Grammar	Automaton
Set of all languages	-	-
Type 0	unrestricted	Turing Machine
Type 1	context-sensitive / monotonic	nondeterministic linear bounded automaton
Type 2	context-free	nondeterministic pushdown automaton
deterministic context-free	LR(k)	deterministic pushdown automaton
Type 3	regular	finite automaton

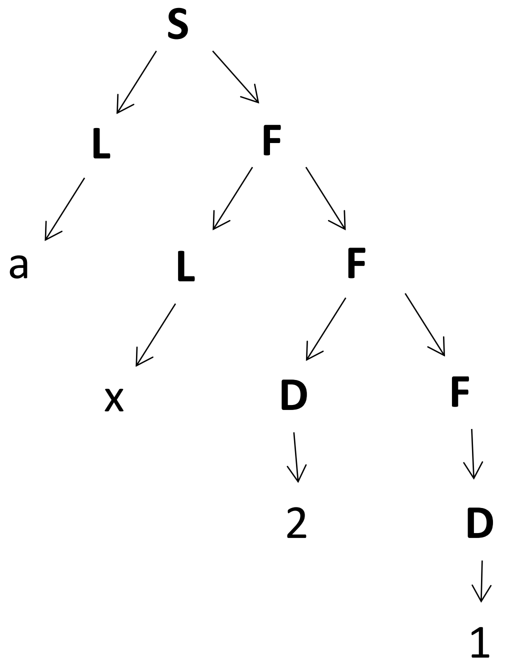
Example: Identifiers in Programming Languages

Grammar for identifiers in the C programming language (for variables, function names)

- String consisting of
  - Latin letters
  - Underscore (_)
  - Decimal digits
- The first character can only be a letter or an underscore
- Grammar:
  V = {**S**, **F**, **L**, **D**}, start symbol **S**
  Σ = {a, b, ..., z, A, B, ..., Z, _, 0, 1, ..., 9}
  P = { **S** ⟶ **L** | **LF**,
  F ⟶ **D** | **L** | **DF** | **LF**,
  L ⟶ a | b | ... | Z | _,
  D ⟶ 0 | 1 | ... | 9 }
- To avoid confusion: nonterminals in bold, terminals printed normally

Example: Identifiers – Syntax Tree

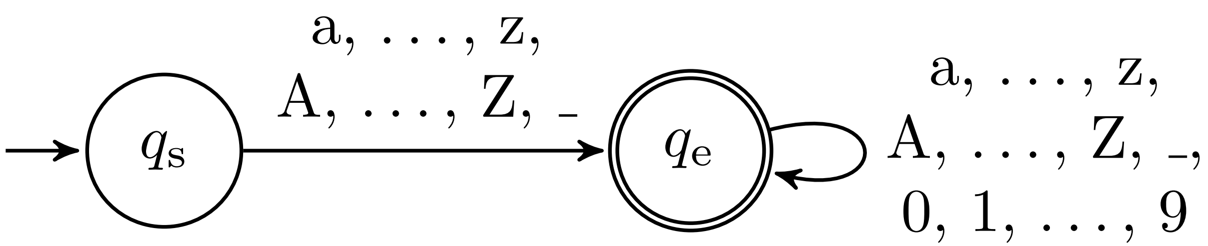
- Derivation of the word ax21
  S ⇒ **LF** ⇒ a**F** ⇒ a**LF** ⇒ ax**F** ⇒ ax**DF** ⇒ ax2**F** ⇒ ax2**D** ⇒ ax21
- Representation as syntax tree:
  - Root = start symbol
  - Number of children a node = word length of the right side of the applied production



V = {**S**, **F**, **L**, **D**}, start symbol **S**
Σ = {a, b, ..., z, A, B, ..., Z, _, 0, 1, ..., 9}
P = {
S ⟶ **L** | **LF**,
F ⟶ **D** | **L** | **DF** | **LF**,
L ⟶ a | b | ... | Z | _,
D ⟶ 0 | 1 | ... | 9 }

Example: Identifiers – Automaton

Representation as a deterministic finite automaton



- Obviously, the language is regular (type 3)
- However, the grammar we used was context-free (type 2)

Example: Identifiers – Regular Grammar

- Type 3 grammar productions, right-linear:

  S ⟶ a | b | … | Z | _
  S ⟶ a**F** | b**F** | … | Z**F** | _**F**
  F ⟶ a | b | … | Z | _ | 0 | 1 | … | 9
  F ⟶ a**F** | b**F** | … | Z**F** | _**F** | 0**F** | 1**F** |… 9**F**

- Notes
  - There is no restriction on the length of the identifiers
  - Length restriction with a finite automaton is only possible, if a separate end state is introduced for each permissible length
  - Using a PDA, however, this problem could easily be solved by keeping record of the length of the identifier on the stack.

Example: A Type 1 Language

$L = \{a^n b^n a^n \mid n \in \mathbb{N}\}$

$V = \{S, A, B\}$, start symbol S
$\Sigma = \{a, b\}$

$$
\begin{aligned}
P = \{ \quad & S \longrightarrow aba \mid aSA \mid a^2bBa \\
& BA \longrightarrow bBa \\
& aA \longrightarrow Aa \\
& B \longrightarrow ba \quad \}
\end{aligned}
$$

- The grammar is monotonic and therefore defines a type 1 language (we will see how to prove that this language is not type 2 later)
- the rule $aA \longrightarrow Aa$ is not context-sensitive, therefore the grammar is type 0

Example: A Type 1 Language – Dead-End Derivations

- Derivation of the word $a^4b^4a^4$:
  $S \Rightarrow aSA \Rightarrow aaSAA \Rightarrow a^2a^2bBaAA \Rightarrow a^4bBAaA \Rightarrow a^4b^2BaaA \Rightarrow a^4b^2BaAa \Rightarrow a^4b^2BAa^2 \Rightarrow$
  $a^4b^2bBaa^2 \Rightarrow a^4b^3baa^3 \Rightarrow a^4b^4a^4$
- or alternatively:
  $S \Rightarrow aSA \Rightarrow aaSAA \Rightarrow a^2a^2bBaAA \Rightarrow a^4bBAaA \Rightarrow a^4bBAAa \Rightarrow a^4bbBaAa \Rightarrow a^4b^2BAa^2 \Rightarrow$
  $a^4b^2bBaa^2 \Rightarrow a^4b^3baa^3 \Rightarrow a^4b^4a^4$
- In type 1 (and type 0) grammars, we can have dead-end derivations (*Sackgassen*):
  A derivation stops before we reach a string containing terminal symbols only
  $S \Rightarrow aSA \Rightarrow a^3bBaA \Rightarrow a^3bbaaA \Rightarrow a^3b^2aAa \Rightarrow a^3b^2Aa^2$

$S \longrightarrow aba \mid aSA \mid a^2bBa$
$BA \longrightarrow bBa$
$aA \longrightarrow Aa$
$B \longrightarrow ba$

Closure (*Abgeschlossenheit*)

- Rules (operations) for combining languages
- Let L_1, L_2, L be formal languages
- Typical operations:
  - Intersection: $L_1 \cap L_2 = \{w \mid w \in L_1 \text{ and } w \in L_2\}$
  - Union: $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$
  - Complement: $\bar{L} = \{w \mid w \in \Sigma^* \text{ without } L\}$
  - Concatenation: $L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$
  - Kleene star: $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$

- A class of formal languages is said to be closed (*abgeschlossen*) under an operation if the resulting language belongs to the same class as the original language(s).

Example: Closure

Type 2 languages are not closed under intersection

- Consider the type 2 languages
  - $L_1 = \{ a^i b^k c^k \mid i, k > 0\}$
  - $L_2 = \{ a^i b^i c^k \mid i, k > 0\}$
- Intersection: $L_1 \cap L_2 = \{ a^i b^i c^i \mid i > 0\}$
  - this language is type 1 as discussed in the previous example

Closure Properties

Language	Intersection	Union	Complement	Concatenation	Kleene Star
Type 3	yes	yes	yes	yes	yes
det.cf.	no	no	yes	no	no
Type 2	no	yes	no	yes	yes
Type 1	yes	yes	yes	yes	yes
Type 0	yes	yes	no	yes	yes

Regular Expressions

Regular Expressions

- Yet another way to describe strings or words of a language
- Languages describable by regular expressions are exactly the regular languages
- They are therefore equivalent to finite automata and type 3 grammars
  - use whichever model is most convenient

String concatenation

Kleene plus: repeat c as often as you like, but at least once

- Example: $abc^{+}(a \mid b)^{*}$

Kleene closure: repeat term as often as you like, you may even omit it completely.
Here: Generates an arbitrarily mixed string of a and b

Parentheses: Define precedence

Or-operator: use either a or b

Regular Expressions – Definition

Σ: Alphabet

Syntax

- each symbol of the alphabet is a regular expression
- the empty set $\emptyset$ is a regular expression
- the empty string ε is a regular expression
- if a and b are regular expressions, then also
  - (a) (Parentheses)
  - ab (Concatenation)
  - (a | b) (Or-operator)
  - a* (Kleene closure)
  - a^+ (Kleene plus – unnecessary, as $a^+ = aa^*$)

Semantics

- $L(x \in \Sigma) = \{x\}$
- $L(\emptyset) = \emptyset$
- $L(\varepsilon) = \{\varepsilon\}$

- $L((a)) = L(a)$
- $L(ab) = \{uv \mid u \in L(a) \text{ and } v \in L(b)\}$
- $L((a \mid b)) = L(a) \cup L(b)$
- $L(a^*) = L(a)^*$

Regular Expressions – Rules

Let a, b, c be regular expressions. The following rules apply:

- Hull-operators ($*$, $^+$) have precedence over multiplication (concatenation) over addition (Or: |)

• $(a \mid b) \mid c = a \mid (b \mid c)$	$(ab)c = a(bc)$	(Associativity)
• $a \mid b = b \mid a$		(Commutativity)
• $a \mid \emptyset = \emptyset \mid a = a$	$a\varepsilon = \varepsilon a = a$	(Neutral elements)
•	$a\emptyset = \emptyset a = \emptyset$	(Annihilation)
• $a \mid a = a$		(Idempotence)
• $a(b \mid c) = ab \mid ac$ $(b \mid c)a = ba \mid ca$		(Distributivity)
• $(a^*)^* = a^*$ $\varepsilon^* = \varepsilon$ $\emptyset^* = \varepsilon$ $a^+ = aa^* = a^*a$ $a^* = \varepsilon \mid a^+$ $(a \mid b)^* = (a^*b^*)^*$		(Hull laws)

Regular Expressions – Exercise

Construct a finite automaton that recognizes the language defined by the following regular expression:

$a^+ (ba \mid b)^* c \mid b$

Regular Expressions – Applications

- Used in compilers to check whether a string is formed syntactically correct (e.g., identifiers)
- Description or validation of semantic properties of strings is not possible with regular expressions

- Other areas of application
  - Word processing: Find, replace, and modify according to patterns
  - Unix/Windows shell
  - Part of some programming languages, e.g., PHP, Perl, Python

- Caution:
  - the constructs shown on the following slides are an exemplary selection
    - they differ, depending on the language/tool used
    - some may not be available
    - usually, additional ones are available
  - often, constructs are available that go beyond the power of regular expressions
    - these are then no longer regular expressions in the sense of theoretical computer science
    - nevertheless, in practice they are unfortunately referred to as such
  - Denoted often as *Regex* or *Regexp* (derived from "regular expression")
    - we will use the term *Regex* here to distinguish these programming constructs from regular expressions as used in theoretical computer science

Regex – Typical Constructs

Symbol	Meaning
^	at the beginning of a string
$	at the end of a string
.	any character
a?	a is optional
a*	no or multiple occurrence of a
a+	one or more occurrences of a
a{2}	a occurs exactly twice
a{3,}	a occurs at least 3 times or more
a{4,11}	a occurs at least 4, maximum 11 times

Symbol	Meaning
()	Parentheses for expressions
(a | b)	Either a or b
[1-6]	a digit between 1 and 6
[d-g]	a lowercase letter between d and g
[E-H]	a capital letter between E and H
[^a-z]	no occurrence of lowercase letters between a and z
[_a-zA-Z]	an underscore and any letter of the alphabet
\s	whitespace
\	Escape symbol, to mask reserved symbols, e.g., \? for ?, or \n for newline

Regex – Example: Decimal Fractions

- Verbal description
  - first symbol: minus or plus sign (optional): [- | \+]?
  - followed by at least one digit: [0-9]+
  - after that there can be a decimal point: \.
  - if this is the case, any number of digits can follow, but at least one: [0-9]+
- Complete regex: [-|\+]?[0-9]+(\.[0-9]+)?

Regex – Example: Strings and Natural Numbers

Technische Hochschule Rosenheim

- Description of a string consisting of
  - natural numbers concatenated with words with any number of latin letters, or vice versa
  - where these strings are separated by spaces
- Regex: ^([a-zA-Z]+|[1-9][0-9]*)(\s([a-zA-Z]+|[1-9][0-9]*))*$
- Note:
  - ^ and $ do not mark the beginning and end of the string in the sense of quotation marks „…“
  - but:
    - ^ means: Match only if the subsequent expression occurs at the beginning of a string
    - $ means: Match only if the previous expression occurs at the end of a string
  - Here:
    - Abcd generates a match
    - öä:Abcd does not generate a match (if you would omit ^, then you would also have a match)

Pumping Lemma

Introduction

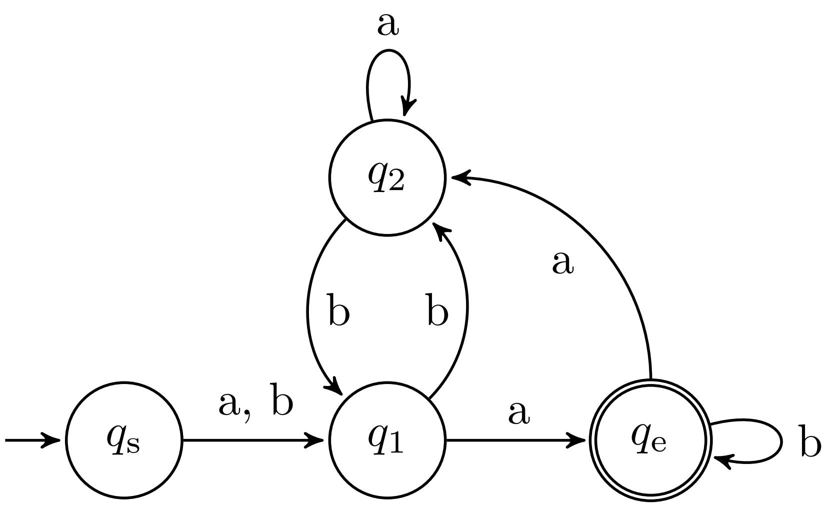
Technische Hochschule Rosenheim

- Pumping Lemma
  - important statement for regular grammars (and thus for finite automata)
  - can be used for many further statements and proofs about regular languages
  - especially useful if you want to show that a language is **not** regular

- a similar theorem exists for context-free languages

Example: What is Pumping?

- let w a word from a regular language
- if w is sufficiently long, it can always be assembled from three parts: w = xyz
- „Pumping“ means: Multiplication of y, e.g., w‘ = xyyz, w‘‘ = xyyyz, …
- this must be possible because:
    - every finite automaton with infinitely large language must go through cycles
    - therefore, repetitions, must occur in the words



- Word w = aa
    - belongs to the language
    - but is too short to pump
- Word w = abba
    - is also part of the language
    - can be pumped
    - with x = a, y = bb, z = a we get w‘ = abbbba, w‘‘ = abbbbbba, …

Pumping Lemma for Regular Languages

- Let L be a regular language
- Then there is a constant n,
  so that each word $w \in L$, with $|w| \geq n$
  can be split into $w = xyz$ with
  - $|xy| \leq n$
  - $|y| \geq 1$
  - $|z|$ arbitrary (can be zero)
- It holds: $x\,y^i\,z \in L$, for all $i = 0, 1, 2, \ldots$

Example: Palindromes Are Not Regular

Proposition: The language L = {w | w is a palindrome over Σ} with Σ = {a, b} is not regular

Proof by contradiction (*Beweis durch Widerspruch*)

- **Assumption**: L is regular, therefore the pumping lemma holds
- w = a^nbaaba^n = xyz is a palindrome from L of sufficient length
- xy can contain only a, because |xy| ≤ n
- In particular, y then contains at least one a
- Pumping lemma: xyyz must also be a word from L
- But: xyy contains at least one a more than the right end of the word, i.e.
  xyyz = a^m baab a^n where m > n
- xyyz is not a palindrome; but it must be, we have to be able to pump: Contradiction!
- **Conclusion: L is not regular**
  - and there exists no finite automaton that only accepts palindromes

This is the n from the pumping lemma!
The minimum (unknown) word length
for pumping.

Some Other Languages that are Not Regular

Using the pumping lemma, e.g., the following languages can be proven to be not regular:

- $L = \{a^n\, b^n \mid n \in \mathbb{N}\}$
  - this language is actually context-free
- $L = \{0^q \mid q \text{ is a square number}\}$
- $L = \{0^p \mid p \text{ is a prime number}\}$
  - And therefore:
    There exists no finite automaton that can decide for a given number whether it is square or prime.

Pumping Lemma for Context-free Languages

- Let L be a context-free language
- Then there is a constant n,
  so that each word $w \in L$, with $|w| \geq n$
  can be split into w = uvxyz with
  - $|vxy| \leq n$
  - $|vy| \geq 1$
  - $|u|$, $|x|$, $|z|$ arbitrary (can be zero)
- It holds: $u\, v^i\, x\, y^i\, z \in L$, for all i = 0, 1, 2, …

Some Other Languages that are Not Context-free

Using the pumping lemma, the following languages can be proven to be not context-free:

- $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$
  - this language actually is context-sensitive
- $L = \{0^q \mid q \text{ is a square number}\}$
- $L = \{0^p \mid p \text{ is a prime number}\}$
  - And therefore: There exists no pushdown automaton that can decide for a given number whether it is square or prime.

Pumping Lemma – Final Notes

- The pumping lemma can only be used to show that a language is not regular/context-free
- We cannot use it to show that it is regular/context-free
- There are languages that meet the pumping theorem but are not regular/context-free

- Example:
  - $L = \{ a^i b^k c^k \mid i, k > 0 \} \cup \{ b^j c^k \mid j,k \geq 0 \}$
  - fulfills the pumping theorem for regular languages
  - but is not regular

Summary

- Grammar: Terminal & nonterminal symbols, production rules
- Chomsky hierarchy of grammars & languages:
  Types 0 (unrestricted) to 3 (most restricted)
- Each type of language has a specific type of automaton & grammar attached to it
- Pumping lemma to prove that a language is not regular/context-free

Sources

- H. Ernst, J. Schmidt und G. Beneken: *Grundkurs Informatik*. Springer Vieweg, 7. Aufl., 2020.
- Schöning, U.: *Theoretische Informatik – kurz gefasst*. Spektrum Akad. Verlag (2008)
- Sander P., Stucky W., Herschel, R.: *Automaten, Sprachen, Berechenbarkeit*, B.G. Teubner, 1992
- D.W. Hoffmann. *Theoretische Informatik*. Hanser, 4. Aufl., 2018.