Theoretische Informatik III (T3INF2002)

Formale Sprachen und Automaten | Einführung Compilerbau

Vorlesung im Wintersemester 2022/23

Formale Sprachen und Automaten

— Reguläre Ausdrücke

Reguläre Ausdrücke

Definition regulärer Ausdrücke

- Alphabet Z = { |, (,), *,Ø } von „Hilfssymbolen“
  („|“ -> „oder“, “*“ -> 0-n Wiederholungen)

- Alphabet A enthalte kein Zeichen aus Z

- Regulärer Ausdruck über A ist eine Zeichenfolge über dem Alphabet A ∪ Z , die gewissen Vorschriften genügt

Reguläre Ausdrücke

- Reguläre Sprachen lassen sich mithilfe regulärer Ausdrücke beschreiben
- Formal sind diese folgendermaßen definiert:

Mit Σ sei ein beliebiges Alphabet gegeben. Reg_Σ, die Menge der regulären Ausdrücke über Σ, wird durch die folgenden Regeln gebildet:

- $\emptyset, \varepsilon \in Reg_\Sigma$
- $\Sigma \subset Reg_\Sigma$
- Mit $r \in Reg_\Sigma$ und $s \in Reg_\Sigma$ sind auch rs und $(r|s) \in Reg_\Sigma$
- Mit $r \in Reg_\Sigma$ sind auch (r) und $r^* \in Reg_\Sigma$

Reguläre Grammatiken und reguläre Ausdrücke

$L_1 := \{(ab)^n \mid n \in \mathbb{N}^+\}$

S -> aB

B -> bC

C -> ε | aB

ab(ab)*

(* -> 0..?-fache Wiederholung)

Reguläre Grammatiken und reguläre Ausdrücke

$L_2 := \{a^i b^j c^k \mid i, j, k \in \mathbb{N}^+\}$

S -> aS | aB

B -> bB | bC

C -> cC | c

aa*bb*cc*

(* -> 0..?-fache Wiederholung)

Nicht-reguläre Sprache

- Nicht jede Sprache lässt sich durch regulären Ausdruck beschreiben
- Selbst wenn eine Sprache auf einem regelmäßig Ansatz aufgebaut ist, heißt das nicht unbedingt, dass sie eine reguläre Sprache ist
- Regulär meint, dass die Sprache durch endlich viele Anwendungen der Operationen *Vereinigung*, *Verkettung* und *Abschluss* auf Elementarsprachen zustande kommt
  - Elementarsprache ist eine Sprache, die nur aus einbuchstabigen Wörtern besteht und auch leer sein kann

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

Wörter in L = {ab, aabb, aaabbb, aaaabbbb, ...}

Nicht-reguläre Sprache

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

Wörter in L = {ab, aabb, aaabbb, aaaabbbb, …}

- Sprache ist zwar regelmäßig aufgebaut, es gibt jedoch keinen regulären Ausdruck, der L beschreiben kann
- Wörter enthalten gleichviele a´s und b´s, was nicht mit einem regulären Ausdruck ausgedrückt werden kann

Nicht-reguläre Sprache

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

Wörter in L = {ab, aabb, aaabbb, aaaabbbb, …}

- Der reguläre Ausdruck a*b* beschreibt zwar Wörter von L, beinhaltet jedoch auch Wörter, die nicht in die Sprache gehören -> *aabbbb*

Reguläre Grammatiken und reguläre Ausdrücke

- Es lässt sich zeigen, dass zu jeder regulären Grammatik ein äquivalenter regulärer Ausdruck existiert und umgekehrt
- Zwischen regulären Grammatiken und regulären Ausdrücken besteht nur ein äußerlicher Unterschied
  - beide erzeugen dieselbe Sprachklasse L_3
- Zusammenhang wird mit der Einführung des endlichen Automaten ersichtlich und es zeigt sich, wie sich reguläre Grammatiken und reguläre Ausdrücke eins zu eins auf den Automatenbegriff reduzieren lassen

Reguläre Grammatiken und reguläre Ausdrücke

- Reguläre Ausdrücke besitzen eine große Bedeutung in der praktischen Informatik
- Sie werden von Kommandozeilenwerkzeugen z. B. für die Spezifikation von Suchmustern verwendet

Bespiele regulärer Ausdrücke

Sei Alphabet A = {a, b}

⇒ gültige reguläre Ausdrücke

∅	a	b
(ab)	(∅ b)	(a∅)
((ab)a)	(((ab)a)a)	((ab)(aa))
(∅|b)	(a|b)	(a|(ab))
((a(a|b))|b)	(a|(b|(a|a)))	

Klammereinsparungsregeln

- „Stern- vor Punktrechnung“
- „Punkt- vor Strichrechnung“

Beispiel:

- $R_1 | R_2R_3^*$ Kurzform für $(R_1 | (R_2(R_3^*)))$

Klammereinsparungsregeln

- Bei mehreren gleichen binären Operatoren ohne Klammern gilt das als links geklammert

Beispiel:

- $R_1 \mid R_2 \mid R_3$ Kurzform für $((R_1 \mid R_2) \mid R_3)$

Beispiele für Klammereinsparungsregeln

abaa	**statt**	*(((ab)a)a)*
ab(aa)	**statt**	*((ab)(aa))*
a(a|b)|b	**statt**	*((a(a|b))|b)*
*(abb)**|ø**	**statt**	*((((ab)b)*)*)|(ø*))*

Beispiel für keine regulären Ausdrücke

keine regulären Ausdrücke über {a, b}:

(|b)	vor „|“ fehlt ein regulärer Ausdruck
|∅|	vor und hinter „|“ fehlt ein regulärer Ausdruck
()ab	zwischen „(“ und „)“ fehlt ein regulärer Ausdruck
((ab)	Klammern müssen paarweise vorhanden sein
***(ab)**	vor „*“ fehlt ein regulärer Ausdruck
c*	c ist kein Zeichen des Alphabetes

Syntax regulärer Ausdrücke

— Alternative Definition mit Hilfe einer kontextfreien Grammatik

$$G = (\{ R \} , \{ | , (,) , * , \emptyset \} \cup A , R , P)$$

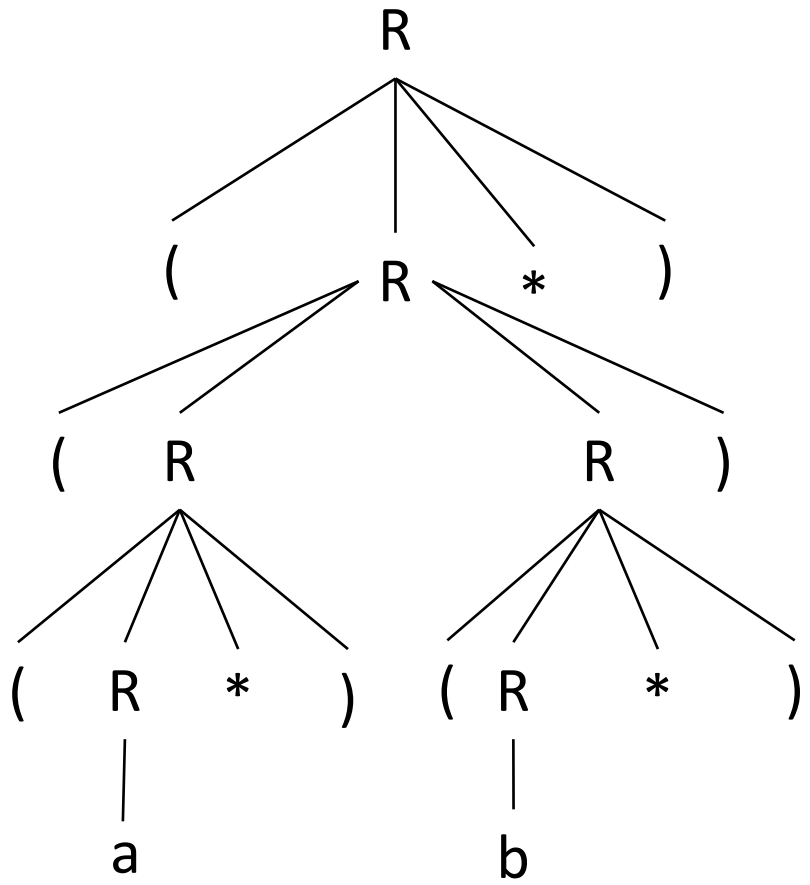
mit $P = \{ R \rightarrow \emptyset ,$

$R \rightarrow x \mid x \in A ,$

$R \rightarrow (R \mid R),\ R \rightarrow (RR) ,$

$R \rightarrow (R^*) \}$

Syntaxbaum (Ableitungsbaum) eines regulären Ausdrucks



$G = (\{ R \}, \{ |, (,), *, \emptyset \} \cup A, R, P)$

mit $P = \{ R \rightarrow \emptyset,$
$R \rightarrow x \mid x \in A,$
$R \rightarrow (R \mid R),\ R \rightarrow (RR),$
$R \rightarrow (R^*) \}$

(((a*)(b*))*) => in kurz: (a*b*)*

Semantik regulärer Ausdrücke

- Eine durch R beschriebene formale Sprache <R> ist wie folgt definiert:

$$\langle \emptyset \rangle = \{\}$$

$$\langle x \rangle = \{x\} \text{ für jedes } x \in A$$

$$\langle R1R2 \rangle = \langle R1 \rangle \cdot \langle R2 \rangle$$

$$\langle R1 \mid R2 \rangle = \langle R1 \rangle \cup \langle R2 \rangle$$

$$\langle R^* \rangle = \langle R \rangle *$$

Beispiel für die formale Sprache <R>

R = a|b dann ist <R> = <a | b> = <a> ∪ <b> = {a} ∪ {b} = {a,b}

Beispiel für die formale Sprache <R>

R = a|b dann ist <R> = <a | b> = <a> ∪ <b> = {a} ∪ {b} = {a,b}

R = $(a|b)^*$ dann ist <R> = $<(a|b)^*>$ = $<a|b>^*$ = $\{a, b\}^*$

Beispiel für die formale Sprache <R>

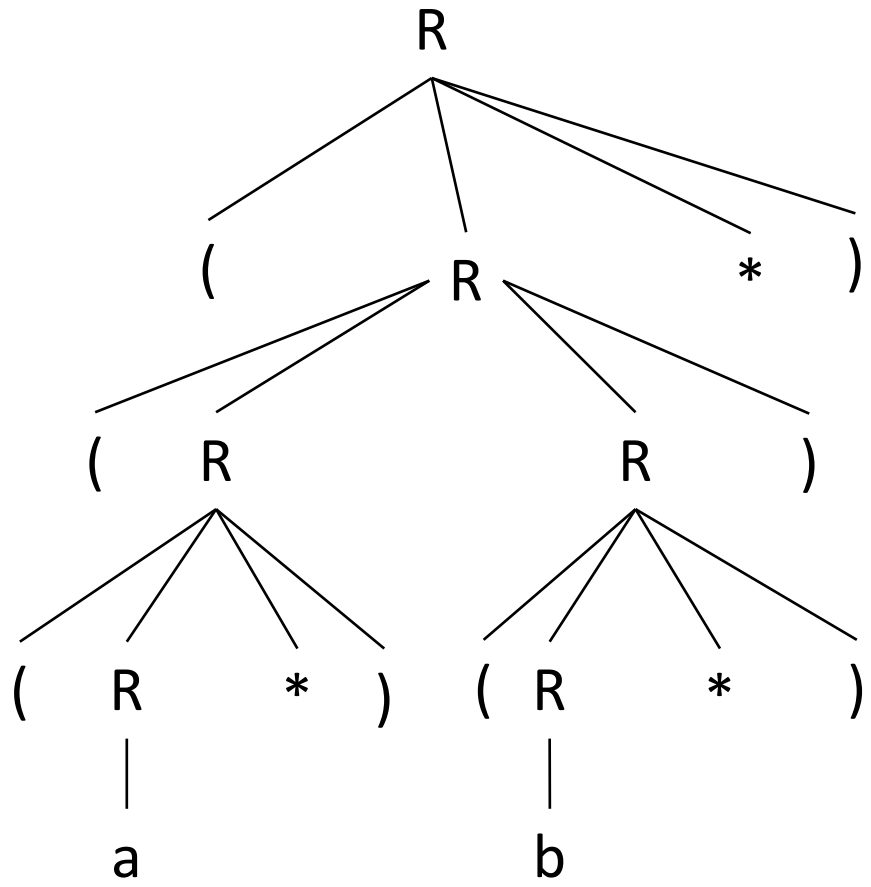
R = a|b dann ist <R> = <a | b> = <a> ∪ <b> = {a} ∪ {b} = {a,b}

R = $(a|b)^*$ dann ist <R> = $<(a|b)^*>$ = $<a|b>^*$ = $\{a, b\}^*$

R = $(a^*b^*)^*$ dann ist <R> = $<(a^*b^*)^*>$ = $<a^*b^*>^*$

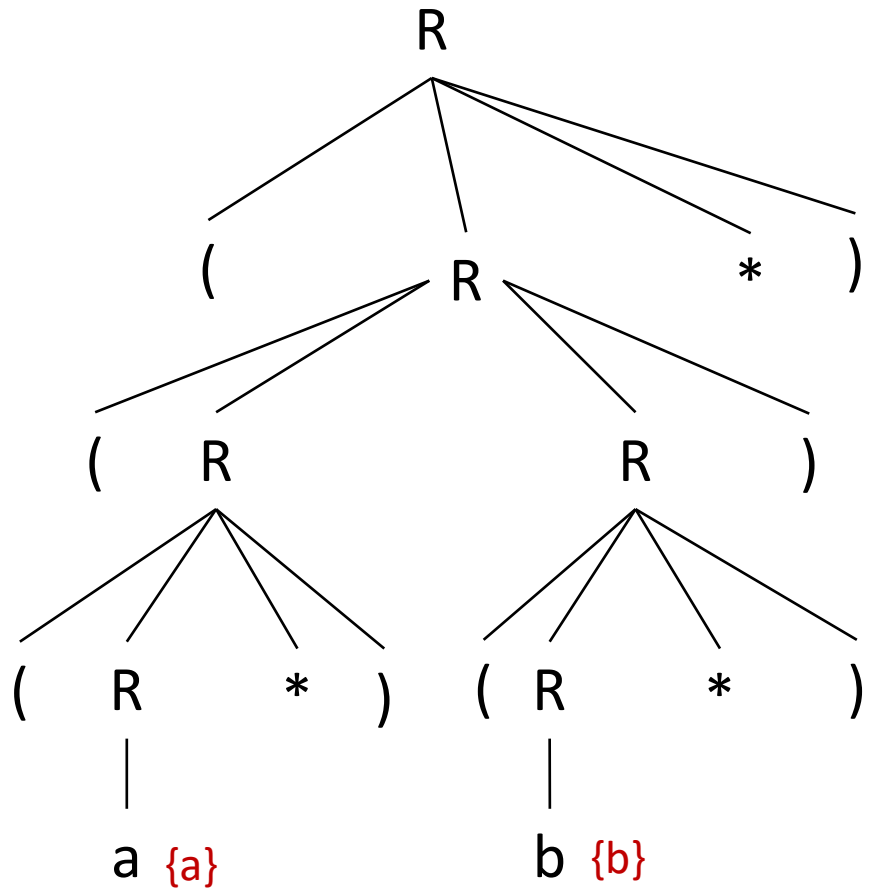
= $(<a^*><b^*>)^*$ = $(<a>^*<b>^*)^*$ = $(\{a\}^*\{b\}^*)^*$

Syntaxbaum (Ableitungsbaum) eines regulären Ausdrucks



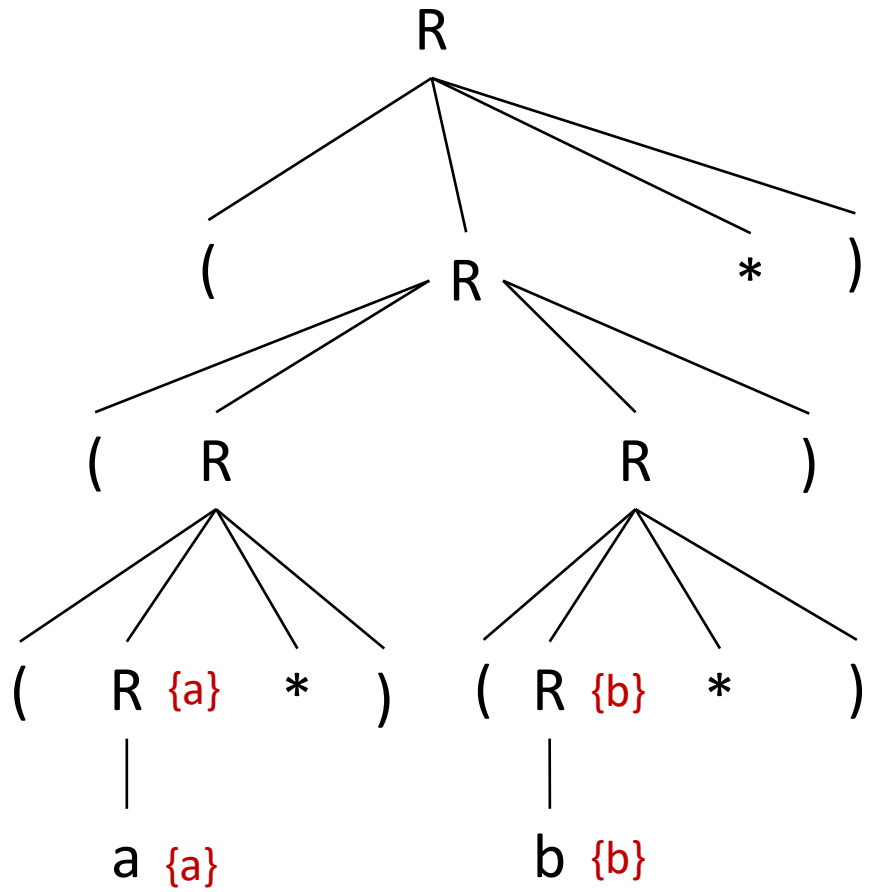
R = (a*b*)* dann ist <R> = <(a*b*)*> = <a*b*>∗
= (<a*><b*>)∗ = (<a>∗<b>∗)∗ = ({a}∗{b}∗)∗

Syntaxbaum (Ableitungsbaum) eines regulären Ausdrucks



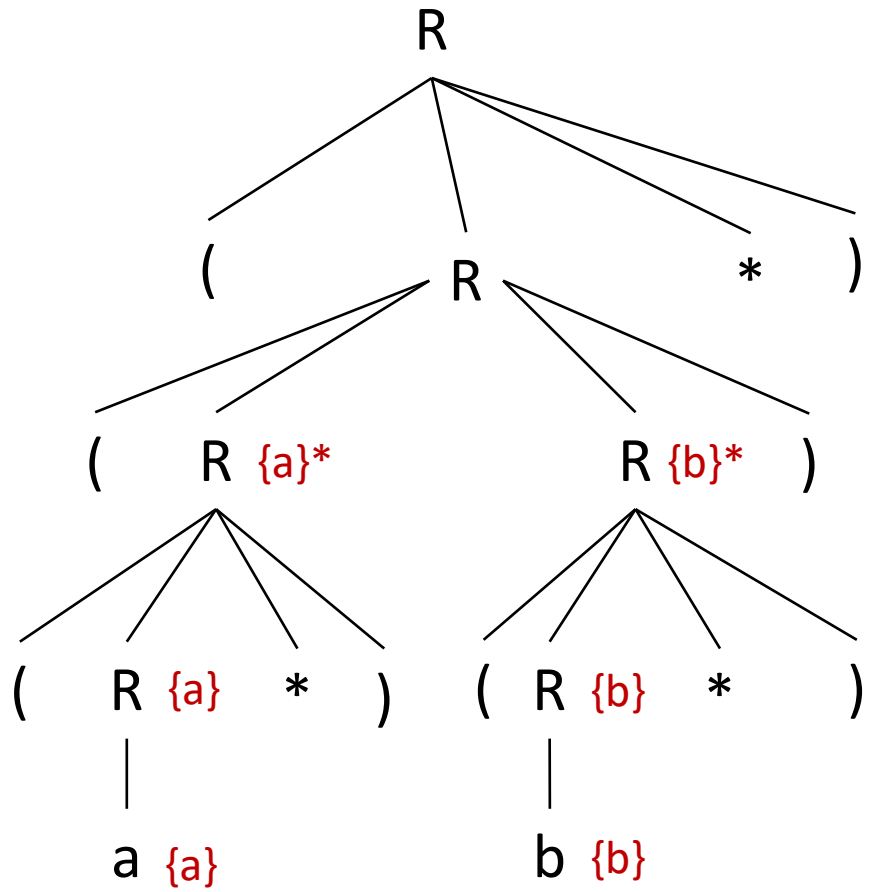
R = (a*b*)* dann ist <R> = <(a*b*)*> = <a*b*>∗
= (<a*><b*>)∗ = (<a>∗<b>∗)∗ = ({a}∗{b}∗)∗

Syntaxbaum (Ableitungsbaum) eines regulären Ausdrucks



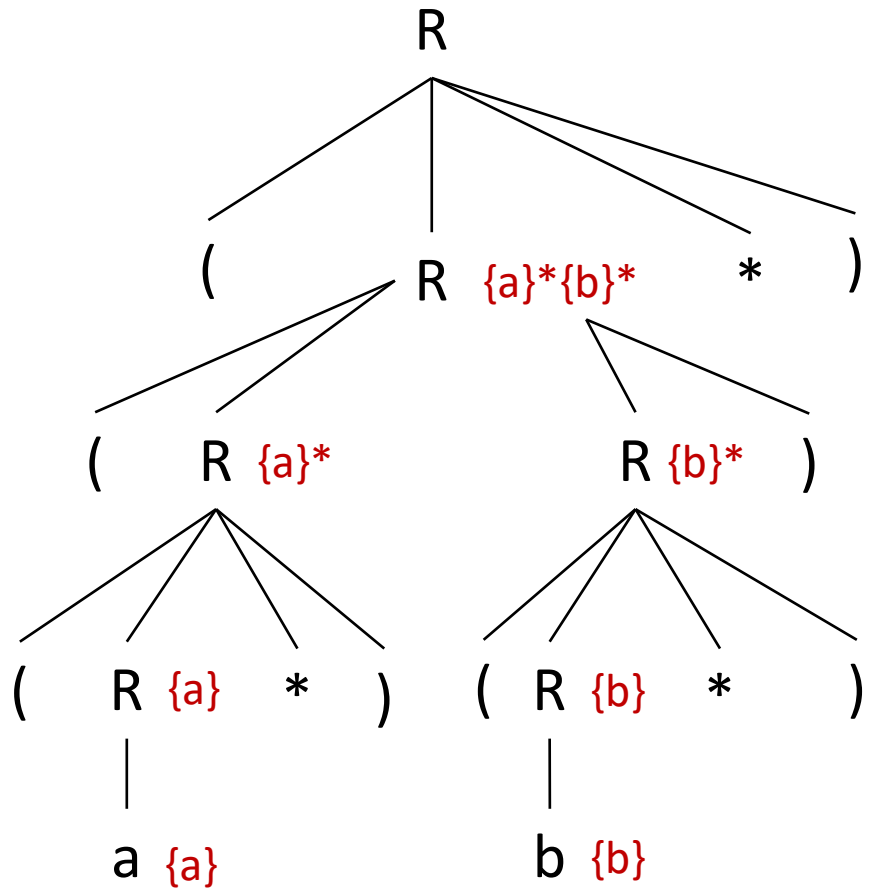
R = (a*b*)* dann ist <R> = <(a*b*)*> = <a*b*>∗
= (<a*><b*>)∗ = (<a>∗<b>∗)∗ = ({a}∗{b}∗)∗

Syntaxbaum (Ableitungsbaum) eines regulären Ausdrucks



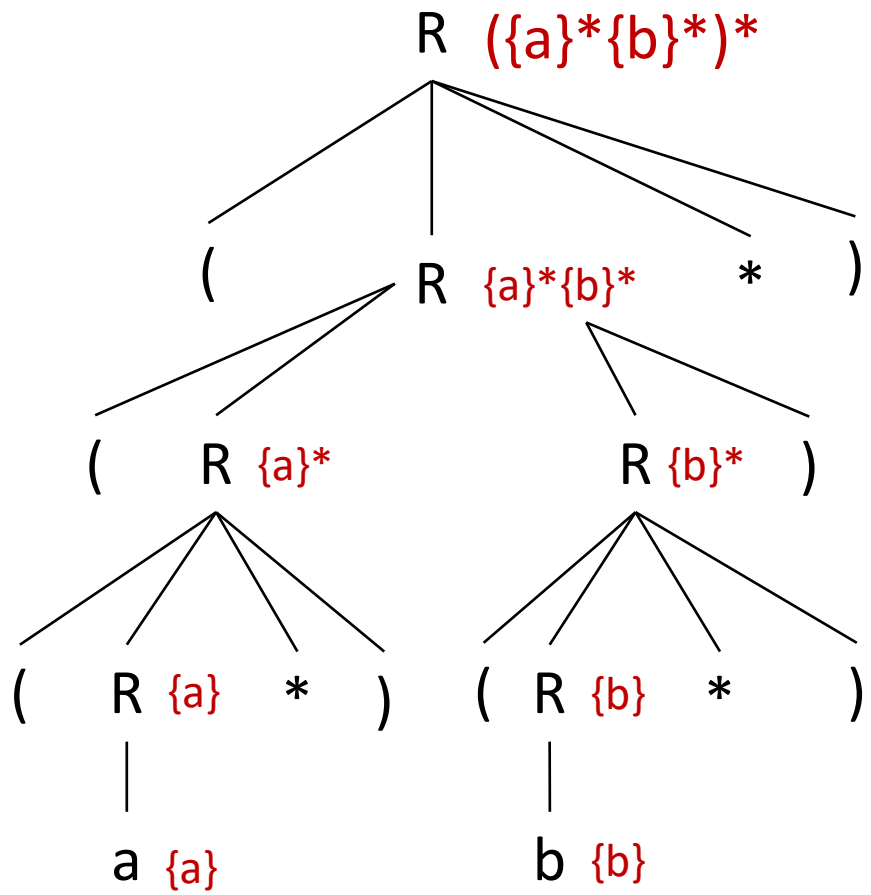
R = (a*b*)* dann ist <R> = <(a*b*)*> = <a*b*>∗
= (<a*><b*>)∗ = (<a>∗<b>∗)∗ = ({a}∗{b}∗)∗

Syntaxbaum (Ableitungsbaum) eines regulären Ausdrucks



R = (a*b*)* dann ist <R> = <(a*b*)*> = <a*b*>∗
= (<a*><b*>)∗ = (<a>∗<b>∗)∗ = ({a}∗{b}∗)∗

Syntaxbaum (Ableitungsbaum) eines regulären Ausdrucks



R = (a*b*)* dann ist <R> = <(a*b*)*> = <a*b*>∗
= (<a*><b*>)∗ = (<a>∗<b>∗)∗ = ({a}∗{b}∗)∗

Beispiel: Datums- und Zeitangaben

Wed, 21 Jan 2022 16:08:47 +0100

```
date-time     = [ day-of-week "," ] date time [CFWS]
day-of-week   = ([FWS] day-name)
day-name      = "Mon" / "Tue" / "Wed" / "Thu" / …
date          = day month year
day           = ([FWS] 1*2DIGIT FWS)
month         = "Jan" / "Feb" / "Mar" / "Apr" /…
year          = (FWS 4*DIGIT FWS)
time          = time-of-day zone
time-of-day   = hour ":" minute [ ":" second ]
hour          = 2DIGIT
minute        = 2DIGIT
second        = 2DIGIT
zone          = (FWS ( "+" / "-" ) 4DIGIT)
```

Beispiel: Datums- und Zeitangaben

DIGIT = 0|1|2|3|4|5|6|7|8|9, also < DIGIT > = {0,1,2,3,4,5,6,7,8,9}

2DIGIT ist Abkürzung für DIGIT DIGIT , also z. B.

<2DIGIT> = {00,01,02,...,99} also z.B.<hour> = {00,01,02,...,99}

- Wörter in Anführungszeichen stehen für sich
- Eckige Klammern bedeuten, dass ein Teil optional ist

Beispiel: Datums- und Zeitangaben

DIGIT = 0|1|2|3|4|5|6|7|8|9, also < DIGIT > = {0,1,2,3,4,5,6,7,8,9}

2DIGIT ist Abkürzung für DIGIT DIGIT , also z. B.

<2DIGIT> = {00,01,02,...,99} also z.B.<hour> = {00,01,02,...,99}

time-of-day = hour : minute | hour : minute : second

Beispiele: 05:15, 05:15:45

Aber: 9:50 ist nicht syntaktisch korrekt, jedoch 39:71

Reguläre Ausdrücke in der Programmierung

Reguläre Ausdrücke in der Programmierung

- Reguläre Ausdrücke werden in der Programmierung nach verschiedenen Konventionen notiert
  - Lassen sich in mehrere verschiedene “Richtungen" oder "Traditionen" einteilen

Beispiel: Zeichenreihen, die ganze oder Dezimalzahlen in der üblichen deutschen Notation darstellen

- Verbal ließen sich diese etwa so beschreiben:
  1. Am Anfang kann, muss aber nicht, ein Vorzeichen stehen.
  2. Danach kommt mindestens eine Ziffer, vielleicht auch mehrere.
  3. Danach kann ein Komma stehen; dann muss aber mindestens noch eine Ziffer folgen, vielleicht auch mehrere.
- In der Programmierung kann der reguläre Ausdruck wie folgt notieret werden:

```
[+-]?[0-9]+(,[0-9]+)?
```

Reguläre Ausdrücke in der Programmierung

Es fällt auf, dass es offensichtlich zwei Arten von Zeichen gibt:

- gewöhnliche Zeichen (terminale Zeichen), die sich selbst bedeuten, z.B. das Komma, welches nur bedeutet, dass dort ein Komma steht, und
- Metazeichen, die für die Bedeutung des regulären Ausdrucks selbst eine Bedeutung haben

Metazeichen Bedeutung

- **Eckigen Klammern:** die eine Liste von Zeichen umschließen, von denen genau eines an einer bestimmten Stelle vorkommen soll
- **Minuszeichen:** das einen Bereich von Zeichen markiert, die dann nicht alle aufgelistet zu werden brauchen (aber nur wenn es innerhalb einer eckigen Klammer und dort nicht am Rand steht)
- **Fragezeichen:** das anzeigt, dass das letzte Zeichen oder der Inhalt der unmittelbar voranstehenden runden Klammer optional ist, d.h. einmal vorkommen kann, aber nicht muss
- **Pluszeichen:** das anzeigt, dass das letzte Zeichen oder der Inhalt der unmittelbar voranstehenden runden Klammer mindestens einmal vorkommen muss aber auch mehrmals vorkommen darf
- **Runde Klammer:** die dafür sorgt, dass sich ein dahinter stehendes Frage- oder Pluszeichen auf einen längeren Text und nicht nur auf das letzte Zeichen bezieht, analog der Verwendung von Klammern in der Mathematik

Weitere Metazeichen

- **Stern:** der anzeigt, dass das letzte Zeichen oder der Inhalt der unmittelbar voranstehenden runden Klammer ein- oder mehrmals vorkommen kann, aber nicht muss,
- **Punkt:** der für genau ein ganz beliebiges Zeichen steht,
- **Senkrechte Strich:** der Alternativen voneinander trennt, von denen mindestens eine zutreffen muss
- weitere, exotischere Zeichen

Metazeichen

- Außer das Minuszeichen sind Zeichen nur dann Metazeichen, wenn sie außerhalb von eckigen Klammern stehen
- Das kann man dazu verwenden, um beispielsweise eine Klammer als terminales Zeichen zu notieren:
  - man schreibt einfach [(], was, soviel heißt wie "eines der folgenden Zeichen: ("
  - Alternativ kann man auch vor ein Metazeichen einen inversen Schrägstrich setzen, um es auf diese Weise zu maskieren, d.h. an dieser Stelle nur als terminales Zeichen zu verwenden

Vorrangregeln

Die Vorrangregeln sind:

- Eckige Klammern binden stärker als die runden
- Stern, Pluszeichen und Fragezeichen binden stärker als Hintereinanderschreiben, d.h. ab* bedeutet a(b*) und nicht (ab)*
- Senkrechte Striche binden schwächer als Hintereinanderschreiben, d.h. ab|c bedeutet (ab)|c und nicht a(b|c).

Notationen für reguläre Ausdrücke

- Bei den regulären Ausdrücken erlebt man das Dilemma von Normierungen:
  - wird etwas genormt, bevor es in allgemeinem Gebrauch ist, so ist die Norm weltfremd und wird nicht akzeptiert
  - ist es aber in allgemeinem Gebrauch, so ist es zu spät für die Normung
- Bei Software-Schnittstellen, wie bei den regulären Ausdrücken, kann eine spätere Normierungen einen davon abweichenden früher üblichen Gebrauch nachträglich entwerten, was man vermeiden will
- Es wurde versucht, die unterschiedlichen Notationen für reguläre Ausdrücke zu vereinheitlichen, so dass es nicht beliebig viele, sondern im wesentlichen nur drei verschiedene Notationen gibt

Notationen für reguläre Ausdrücke

Hauptproblem bei der Vereinheitlichung der Notation für reguläre Ausdrücke sind die Metazeichen:

- wird die Notation um zusätzliche Operatoren erweitert, so werden Zeichen, die bisher terminale Zeichen waren, plötzlich zu Metazeichen, ändern also in bestehenden Programmen ihre Bedeutung
- Beispielsweise ist das Fragezeichen zur Bezeichnung optionaler Teile erst später dazugekommen, in früheren Programmen gab es das nicht
- Mit dieser Neuerung ändert aber jeder reguläre Ausdruck, der zufällig ein Fragezeichen enthält, seine Bedeutung

Bekannte Notationen

Die verbreitetsten Notationen für reguläre Sprachen:

- ERE: Extended Regular Expression (=erweiterter regulärer Ausdruck) gemäß X/Open CAE Specification "System Interface Definitions", Issue 4
- BRE: Basic Regular Expression (=einfacher regulärer Ausdruck) gemäß dem selben Dokument
- Shell: Wildcards (=Joker) nach den Konventionen von Unix-Shells
- SQL: Wildcards (=Joker) nach den Konventionen von SQL-Datenbankabfragen

Bekannte Notationen

Während die Notationen ERE und BRE relativ ähnlich sind, unterscheiden sich Shell-Namensmuster von eigentlichen regulären Ausdrücken ganz wesentlich in der Art der Darstellung:

- Bei den ersteren gibt es ein Metazeichen, den Stern, für einen beliebigen (evtl. auch leeren) String, während es bei ERE und BRE ein Metazeichen mit einer solchen Bedeutung nicht gibt
- Der **Stern** bei **ERE** und **BRE** hat seine Bedeutung nur im Zusammenhang mit dem unmittelbar vorhergehenden Zeichen oder geklammerten Teilausdruck und zeigt dessen **Wiederholung** an
- Um einen **beliebigen String** als **ERE** oder **BRE** darzustellen, wird der **Punkt** für ein beliebiges Zeichen mit dem Stern für dessen Wiederholung kombiniert

Bekannte Notationen

Beispiel:

- [ab]* als ERE oder BRE steht für eine beliebige Folge von a und b, z.B. aabbaba; diese Menge erlaubter Strings ist nicht als Shell-Namensmuster darstellbar
- [ab]* als Shell-Namensmuster steht für einen beliebigen String, der mit a oder b beginnt

  -> diese Menge erlaubter Strings würde als ERE oder BRE mit dem Ausdruck [ab].* dargestellt

Auszug zu den Eigenschaften verbreiteter Notationen

	Theorie	ERE	BRE	Wildcard	
				Shell	SQL
leere Menge	Ø	(fehlt)	(fehlt)	(fehlt)	(fehlt)
einzelnes Zeichen	a	a	a	a	a
Konkatenation (Hintereinanderschreiben)	xy	xy	xy	xy	xy
Alternative, Vereinigung	x | y	x | y	(fehlt)	(fehlt)	(fehlt)
Klammer	(. . .)	(…)	\(… \)	(fehlt)	(fehlt)
Rückverweis auf Inhalt der i-ten Klammer	(fehlt)	(fehlt)	\i	(fehlt)	(fehlt)
einzelnes Zeichen aus endlicher Menge	a | b | c	[abc]	[abc]	[abc]	(fehlt)
einzelnes Zeichen nicht aus endlicher Menge	(fehlt)	[^abc]	[^abc]	[!abc]	(fehlt)
beliebiges einzelnes Zeichen	(fehlt)	.	.	?	_
beliebiger Text	(fehlt)	.*	.*	*	%

Shell-Namensmuster

Die Shell (im Folgenden sh, ksh, bash, für tcsh gelten zum Teil in Details abweichende Regeln) verwendet die Namensmuster-Notation (Punkt hat keine spezielle Bedeutung) an folgenden Stellen:

- Vor der Abarbeitung jedes Kommandos werden alle Wörter der Kommandozeile (Kommandoname, Parameter und Operanden) durch eine Liste der Dateipfade ersetzt, die auf das Muster passen
- Wird keine einzige Datei gefunden, so bleibt das Muster erhalten
- Bei der Ersetzung gibt es die Sonderregel, dass ein Punkt, der am Anfang oder nach einem Schrägstrich steht, sowie jeder Schrägstrich explizit angegeben werden muss und nicht durch *, ? oder einen Ausdruck in eckigen Klammern mit abgedeckt ist
- Alternativen in der case-Anweisungen sind Listen von Namensmustern in Shell-Notation
- Sonderregel für Punkte und Schrägstriche gilt hier nicht

Shell-Namensmuster

- Man kann bei der Substitution von Variablennamen durch ihre Werte das Ergebnis dahingehend abändern, dass ein Präfix oder Suffix des Variablenwertes abgeschnitten wird, das auf ein gegebenes Muster passt
- Solche Muster werden dann in Shell-Notation ohne die Sonderregel für Dateipfade geschrieben
- Will man bspw. im Shell-Prompt nur den Namen, nicht aber den ganzen Pfad des aktuellen Verzeichnisses ausgeben, schreibt man ${PWD##*/} und meint damit den Wert ${PWD}, verkürzt um das längste Präfix, das auf das Muster */ passt
- Es gibt dabei vier Möglichkeiten:

%	: verkürze um das kürzeste Suffix, das auf das Muster passt
%%	: verkürze um das längste Suffix, das auf das Muster passt
#	: verkürze um das kürzeste Präfix, das auf das Muster passt
##	: verkürze um das längste Präfix, das auf das Muster passt

Shell-Namensmuster

- Programme haben **find**, **pax** und **cpio** Argumente, bei denen das Programm eine Suche nach Dateien entsprechend einem Muster durchführt, wobei das Muster in Shell-Notation geschrieben wird. Bei **pax** ist ausdrücklich spezifiziert, dass die Sonderregel für Dateipfade wie bei der Dateisubstitution ebenfalls gilt.
- Bei diesen Programmen muss man die Shell daran hindern, eine solche Substitution schon vorab durchzuführen:

  find . -name 'abc' -print*

  sucht nach allen Dateien, deren *name* mit *abc* beginnt und könnte beispielsweise ./abc1, ./abc2 und ./sub/abc als Ergebnis liefern.
- Demgegenüber würde

  find . -name abc -print*

  schon von der Shell durch

  find . -name abc1 abc2 -print

ersetzt, was syntaktisch falsch ist und, wenn es zufällig richtig wäre, nach etwas ganz anderem suchen würde.

Reguläre Ausdrücke

Die meisten Programme, die reguläre Ausdrücke verarbeiten, richten sich nach den Regeln von Extended oder Basic Regular Expressions. Leider haben fast alle diese Programme Besonderheiten, in denen sie von der Norm abweichen. Die Tabelle zeigt einige wenige Beispiele:

Programm	Parameter	ERE/BRE/ Shell-Namensmuster	Besonderheiten
awk		E	Es gibt weitere mit \ eingeleitete Ersatzdarstellungen von Zeichen, etwa \\ für den inversen Schrägstrich selbst und \a, \b, \f, \n, \r, \t und \v für die ASCII-Steuerzeichen BEL, BS, FF, LF, CR, TAB und VT.
cpio	*Operanden*	Shell-Namensmuster	
csplit	*Operanden*	E	
ed		B	Mit dem Schrägstrich (manchmal auch Fragezeichen für Rückwärts-Suche) werden die Suchmuster begrenzt

Maskierung und Mengen von Einzelzeichen

- Ein bestimmtes einzelnes Zeichen wird einfach durch sich selbst dargestellt, es sei denn, dass es innerhalb regulärer Ausdrücke eine besondere Bedeutung hat, wie etwa:
  - der Punkt (beliebiges Zeichen), der Stern (Wiederholung des letzten Teilausdrucks),
  - die öffnende eckige Klammer (Beginn der Notation einer Menge von Einzelzeichen) oder
  - der inverse Schrägstrich (Maskierung, Beginn einer Klammerung).

- In diesem Fall muss es maskiert werden
  - Einfachste Möglichkeit der Maskierung besteht darin, es in eckige Klammern zu setzen; dadurch wird eine Zeichenmenge definiert, die genau dieses eine Zeichen enthält
  - Funktioniert für alle Zeichen, auch die eckigen Klammern selbst, nur nicht für den Zirkumflex-Akzent

Maskierung und Mengen von Einzelzeichen

- Alternative besteht darin, dem zu maskierenden Zeichen einen inversen Schrägstrich voranzustellen; es muss sich dann aber wirklich um ein Zeichen handeln, das sonst speziell behandelt würde, während das bei der Maskierung mit der eckigen Klammer gleichgültig ist
- Gemäß der ERE-Syntax kann man die Sprache *"eines der Zeichen a, b oder c"* als (a|b|c) darstellen, aber spätestens bei der Sprache *"ein Buchstabe aus dem Bereich von a bis z"* wird diese Darstellung unhandlich und man greift zu der abgekürzten Form [a-z]
- Gemäß der BRE-Syntax ist dies sogar die einzige Möglichkeit, da der senkrechte Strich, mit dem Alternativen notiert werden, dort nicht existiert

Maskierung und Mengen von Einzelzeichen

Innerhalb der eckigen Klammer verlieren die meisten Zeichen ihre besondere Bedeutung, die sie sonst in regulären Ausdrücken haben.

Dafür gibt es leider auch Zeichen, die eine besondere Bedeutung erlangen:

- Öffnende eckige Klammer hat eine Sonderbedeutung, wenn ihr ein Punkt, Gleichheitszeichen oder Doppelpunkt folgt
- Schließende eckige Klammer beendet die ganze Liste oder den von einer öffnenden eckigen Klammer begonnenen Teil, es sei denn, sie steht ganz vorne in der Liste
- Mit Bindestrich (Minuszeichen) wird ein Bereich bezeichnet, es sei denn, er steht ganz am Anfang oder am Ende der Liste

Maskierung und Mengen von Einzelzeichen

Man bekommt beim Aufbau der Liste keine Probleme, wenn man die Zeichen wie folgt anordnet:

- Ist ein Bindestrich dabei, so kommt er ans Ende
- Ist eine öffnende eckige Klammer dabei, so kommt sie ans Ende, jedoch vor einen gegebenenfalls ebenfalls vorhandenen Bindestrich
- Ist eine schließende eckige Klammer dabei, so kommt sie an den Anfang
- Ist ein Zirkumflex-Akzent dabei, so kommt er nicht an den Anfang; er verdrängt dabei notfalls Zeichen, die nach den voranstehenden Regeln an das Ende sollen
- Wenn die Liste fertig ist, kommt ein Zirkumflex-Akzent davor, wenn der Ausdruck die Zeichen darstellen soll, die nicht in der Liste vorkommen
- Am Schluss kommen die eckigen Klammern drum herum

Maskierung und Mengen von Einzelzeichen

- Welche Zeichen in einem Bereich liegen, der mit Bindestrich notiert wird, richtet sich nicht nach der Codierung (etwa im ASCII- oder ISO8859-1-Code), sondern nach der Sortierreihenfolge, die systemweit oder benutzerspezifisch nach den Gepflogenheiten einzelner Länder eingestellt sein kann
- Es ist nicht von vorneherein klar, ob das scharfe ß im Bereich [a-z] enthalten ist oder nicht
- Spezifiziert man explizit die "POSIX Locale" durch Setzen der Environment-Variablem LC_COLLATE auf den Wert POSIX, so liegen die Zeichen des ASCII-Codes in der durch diesen Code spezifizierten Reihenfolge; über die übrigen Schriftzeichen sagt die Norm nichts aus

Maskierung und Mengen von Einzelzeichen

Kennt man die Sortierreihenfolge und Zeichenklassen der eingestellten "Locale", so gibt es weitere Darstellungen von Zeichenmengen wie etwa:

- [.ch.] (die Zeichenfolge ch, aber nur falls sie bei der Sortierung gemeinsam als ein Zeichen betrachtet wird)
- [=a=] (die Zeichen, die so wie *a* einsortiert werden)
- [:digit:] (die Zeichen, die lokal als Ziffern betrachtet werden)

Maskierung und Mengen von Einzelzeichen

Zeichenklasse	Bedeutung	internationale Voreinstellung ("POSIX-Locale")
[:alnum:]	alphanumerisches Zeichen	[[:alpha:][:digit:]]
[:alpha:]	Buchstabe	[[:upper:][:lower:]]
[:blank:]	Zwischenraum	*Zwischenraum und horiz. Tabulator*
[:cntrl:]	Steuerzeichen	*Steuerzeichen ausschließlich Zwischenraum*
[:digit:]	Dezimalziffer	[0123456789]
[:graph:]	Schriftzeichen	[[:alpha:][:digit:][:punct:]]
[:lower:]	Kleinbuchstabe	[abcdefghijklmnopqrstuvwxyz]
[:print:]	druckbares Zeichen	[[:graph:]]
[:punct:]	Satzzeichen	[]!"#$%&'()*+,./:;<=>?@\^_`{|}~[-]
[:space:]	Freiraum	*alle Arten Zwischenraum, Zeilenwechsel und Tabulator*
[:upper:]	Großbuchstabe	[ABCDEFGHIJKLMNOPQRSTUVWXYZ]
[:xdigit:]	hexadekadische Ziffer	[[:digit:]A-Fa-f]