Reguläre Sprachen, Ausdrucksstärke

BC George (HSBI)

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.

Motivation

Was muss ein Compiler wohl als erstes tun?

- Syntax überprüfen
- „Wörter" überprüfen
- Token (Lexeme) erkennen

Themen für heute

- Endliche Automaten
- Reguläre Sprachen
- Lexer

Endliche Automaten

Alphabete

Def.: Ein **Alphabet** Σ ist eine endliche, nicht-leere Menge von Symbolen. Die Symbole eines Alphabets heißen *Buchstaben*.

Def.: Ein **Wort** *w über einem Alphabet* Σ ist eine endliche Folge von Symbolen aus Σ. ϵ ist das leere Wort. Die *Länge* $|w|$ eines Wortes w ist die Anzahl von Buchstaben, die es enthält (Kardinalität).

Def.: Eine **Sprache** L *über einem Alphabet* Σ ist eine Menge von Wörtern über diesem Alphabet. Sprachen können endlich oder unendlich viele Wörter enthalten.

Beispiel

Elemente kann man hintereinander hängen

$\Rightarrow$ Sei M eine Menge

M^* die Menge aller (endlich vielen) hintereinander gehängten Elemente,
einschließlich ε

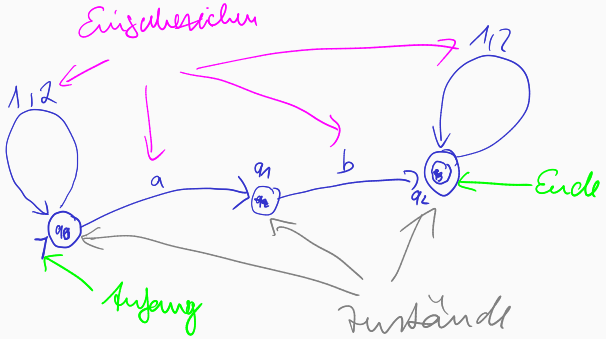
↓

Kleene'sche Hülle

$M^+ : M^*$ ohne $\{\varepsilon\}$

Deterministische endliche Automaten

DFA



state machine

Wenn die Eingabe komplett verarbeitet wurde und der Automat in einem Endzustand ist, wurde das Wort akzeptiert

Def.: Ein **deterministischer endlicher Automat** (DFA) ist ein 5-Tupel $A = (Q, \Sigma, \delta, q_0, F)$ mit

- Q : endliche Menge von **Zuständen**
- Σ : Alphabet von **Eingabesymbolen**
- δ : die (eventuell partielle) **Übergangsfunktion** $(Q \times \Sigma) \to Q, \delta$ kann partiell sein
- $q_0 \in Q$: der **Startzustand**
- $F \subseteq Q$: die Menge der **Endzustände**

Die Übergangsfunktion

Def.: Wir definieren $\delta^* : (Q \times \Sigma^*) \to Q$: induktiv wie folgt:

- Basis: $\delta^*(q, \epsilon) = q \ \forall q \in Q$
- Induktion: $\delta^*(q, a_1, \ldots, a_n) = \delta(\delta^*(q, a_1, \ldots, a_{n-1}), a_n)$

Def.: Ein DFA akzeptiert ein Wort $w \in \Sigma^*$ genau dann, wenn $\delta^*(q_0, w) \in F$.

Beispiel

	a	b
→ q_0	q_1	q_1
q_1	q_0	—

Nichtdeterministische endliche Automaten

Def.: Ein **nichtdeterministischer endlicher Automat** (NFA) ist ein 5-Tupel $A = (Q, \Sigma, \delta, q_0, F)$ mit

- Q : endliche Menge von **Zuständen**
- Σ : Alphabet von **Eingabesymbolen**
- δ : die (eventuell partielle)Übergangsfunktion $(Q \times \Sigma) \to \mathcal{P}(Q)$
- $q_0 \in Q$: der **Startzustand**
- $F \subseteq Q$: die Menge der **Endzustände**

Potenzmenge
↓
Menge aller Teilmenge

Akzeptierte Sprachen

Def.: Sei A ein DFA oder ein NFA. Dann ist **L(A)** die von A akzeptierte Sprache, d. h.

$L(A) = \{\text{Wörter } w \mid \delta^*(q_0, w) \in F\}$

Wozu NFAs im Compilerbau?

Pattern Matching (Erkennung von Schlüsselwörtern, Bezeichnern, ...) geht mit NFAs.

NFAs sind so nicht zu programmieren, aber:

Satz: Eine Sprache L wird von einem NFA akzeptiert $\Leftrightarrow$ L wird von einem DFA akzeptiert.

D. h. es existieren Algorithmen zur

- Umwandlung von NFAs in DFAS
- Minimierung von DFAs (Anzahl d. Zustände)

Reguläre Sprachen

Reguläre Ausdrücke definieren Sprachen

Def.: Induktive Definition von **regulären Ausdrücken** (regex) und der von ihnen repräsentierten Sprache **L**:

- Basis:
  - ϵ und $\emptyset$ sind reguläre Ausdrücke mit $L(\epsilon) = \{\epsilon\}$, $L(\emptyset) = \emptyset$
  - Sei a ein Symbol $\Rightarrow$ a ist ein regex mit $L(a) = \{a\}$
- Induktion: Seien E, F reguläre Ausdrücke. Dann gilt:
  - $E + F$ ist ein regex und bezeichnet die Vereinigung $L(E + F) = L(E) \cup L(F)$
  - EF ist ein regex und bezeichnet die Konkatenation $L(EF) = L(E)L(F)$
  - E^* ist ein regex und bezeichnet die Kleene-Hülle $L(E^*) = (L(E))^*$
  - (E) ist ein regex mit $L((E)) = L(E)$

Vorrangregeln der Operatoren für reguläre Ausdrücke: *, Konkatenation, +

Beispiel

regex	Sprache
01	$\{01\}$
$(a+b)$	$\{a, b\}$
$(a+b)^*$	$\{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$

unendlich viele

jedes endlich lang

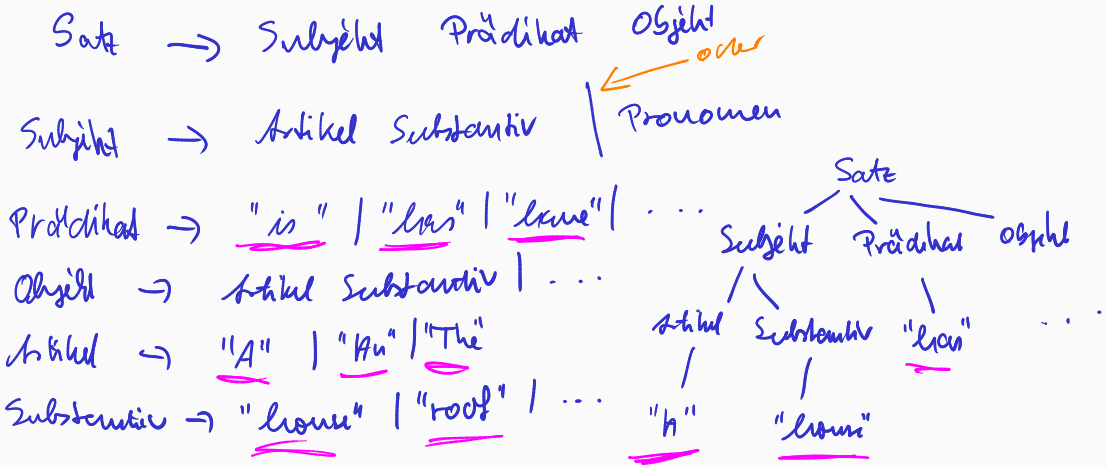
Wichtige Identitäten

Satz: Sei A ein DFA $\Rightarrow \exists$ regex R mit $L(A) = L(R)$.

Satz: Sei E ein regex $\Rightarrow \exists$ DFA A mit $L(E) = L(A)$.

} Algorithmen zur Umwandlung

Formale Grammatiken



Formale Definition formaler Grammatiken

Def.: Eine *formale Grammatik* ist ein 4-Tupel $G = (N, T, P, S)$ aus

- N: endliche Menge von **Nichtterminalen** Subjekt, Prädikat,
- T: endliche Menge von **Terminalen**, $N \cap T = \emptyset$ "Haus", "is" usw.
- $S \in N$: **Startsymbol** Satz
- P: endliche Menge von **Produktionen** der Form

$$X \to Y \text{ mit } X \in (N \cup T)^* N (N \cup T)^*, Y \in (N \cup T)^*$$

Ableitungen

Def.: Sei $G = (N, T, P, S)$ eine Grammatik, sei $\alpha A \beta$ eine Zeichenkette über $(N \cup T)^*$ und sei $A \to \gamma$ eine Produktion von G.

Wir schreiben: $\alpha A \beta \Rightarrow \alpha \gamma \beta$ ($\alpha A \beta$ leitet $\alpha \gamma \beta$ ab).

Def.: Wir definieren die Relation $\overset{*}{\Rightarrow}$ induktiv wie folgt:

- Basis: $\forall \alpha \in (N \cup T)^* \alpha \overset{*}{\Rightarrow} \alpha$ (Jede Zeichenkette leitet sich selbst ab.)
- Induktion: Wenn $\alpha \overset{*}{\Rightarrow} \beta$ und $\beta \Rightarrow \gamma$ dann $\alpha \overset{*}{\Rightarrow} \gamma$

Def.: Sei $G = (N, T, P, S)$ eine formale Grammatik. Dann ist $L(G) = \{\text{Wörter } w \text{ über } T \mid S \overset{*}{\Rightarrow} w\}$ die von G erzeugte Sprache.

Beispiel

$5.0.$

Reguläre Grammatiken

Def.: Eine **reguläre (oder type-3-) Grammatik** ist eine formale Grammatik mit den folgenden Einschränkungen:

- Alle Produktionen sind entweder von der Form
  - $X \to aY$ mit $X \in N, a \in T, Y \in N$ (*rechtsreguläre* Grammatik) oder
  - $X \to Ya$ mit $X \in N, a \in T, Y \in N$ (*linksreguläre* Grammatik)

  exor
- $X \to \epsilon$ ist erlaubt

Beispiel

Satz → "A" A | "The" A | "An" A

A → "house" B | "road" B | ...

B → "is" C | "has" C | ...

C → ...

Reguläre Sprachen und ihre Grenzen

Satz: Die von endlichen Automaten akzeptiert Sprachklasse, die von regulären Ausdrücken beschriebene Sprachklasse und die von regulären Grammatiken erzeugte Sprachklasse sind identisch und heißen **reguläre Sprachen**.

Reguläre Sprachen

- einfache Struktur
- Matchen von Symbolen (z. B. Klammern) nicht möglich, da die fixe Anzahl von Zuständen eines DFAs die Erkennung solcher Sprachen verhindert.

Wozu reguläre Sprachen im Compilerbau?

- Reguläre Ausdrücke if while ; : + ...
  - definieren Schlüsselwörter und alle weiteren Symbole einer Programmiersprache, z. B. den Aufbau von Gleitkommazahlen
  - werden (oft von einem Generator) in DFAs umgewandelt
  - sind die Basis des *Scanners* oder *Lexers* Zerhacker

Lexer

Ein Lexer ist mehr als ein DFA

- Ein **Lexer**
  - wandelt mittels DFAs aus regulären Ausdrücken die Folge von Zeichen der Quelldatei in eine Folge von sog. Token um
  - bekommt als Input eine Liste von Paaren aus regulären Ausdrücken und Tokennamen, z. B. (“while”, WHILE)
  - Kommentare und Strings müssen richtig erkannt werden. (Schachtelungen)
  - liefert Paare von Token und deren Werte, sofern benötigt, z. B. (WHILE, _), oder (IDENTIFIER, “radius”) oder (INTEGERZAHL, “334”)

Bezeichner

Wie geht es weiter?

- Ein **Parser**
  - führt mit Hilfe des Tokenstreams vom Lexer die Syntaxanalyse durch
  - basiert auf einer sog. kontextfreien Grammatik, deren Terminale die Token sind
  - liefert die syntaktische Struktur in Form eines Ableitungsbaums (**syntax tree**, **parse tree**), bzw. einen **AST** (abstract syntax tree) ohne redundante Informationen im Ableitungsbaum (z. B. Semikolons)
  - liefert evtl. Fehlermeldungen

Wrap-Up

Wrap-Up

- Definition und Aufgaben von Lexern
- DFAs und NFAs
- Reguläre Ausdrücke
- Reguläre Grammatiken
- Zusammenhänge zwischen diesen Mechanismen und Lexern, bzw. Lexergeneratoren

LICENSE

Unless otherwise noted, this work is licensed under CC BY-SA 4.0.