Blatt 01; Reguläre Sprachen

A1.1: Sprachen von regulären Ausdrücken

Gegeben ist der reguläre Ausdruck

$$a \;+\; a\,(a+b)^*\,a.$$

Die zugehörige Sprache ist

$$L = \{a\} \cup a\,(a+b)^*\,a.$$

-> Alle Wörter über $\{a, b\}$, die mit a beginnen und mit a enden, einschließlich des einzelnen Zeichens a.

Beispiele: a, aa, aba, $abba$, $aaba$, ...

A1.2: Bezeichner in Programmiersprachen (3P)

In dieser Aufgabe geht es darum, gültige Namen (Bezeichner) für Variablen, Parameter oder andere Elemente einer Programmiersprache zu beschreiben.

Regeln

Ein gültiger Bezeichner muss:

- mit einem Buchstaben beginnen. Bei Variablen ist das `V` oder `v`, bei Parametern `p` oder `P`.
- danach aus Buchstaben (a–z, A–Z), Ziffern (0–9) oder Unterstrichen (_) bestehen,
- *nicht* mit einem Unterstrich enden,
- und mindestens zwei Zeichen lang sein.

(a) Regulärer Ausdruck

Der folgende reguläre Ausdruck beschreibt genau diese gültigen Bezeichner:

```
^[A-Za-z][A-Za-z0-9_]*[A-Za-z0-9]$
```

Das bedeutet:

- `^[A-Za-z]` → Das erste Zeichen muss ein Buchstabe sein.
- `[A-Za-z0-9_]*` → Danach dürfen beliebig viele Buchstaben, Zahlen oder Unterstriche folgen.
- `[A-Za-z0-9]$` → Das letzte Zeichen muss ein Buchstabe oder eine Ziffer sein (kein Unterstrich).

Beispiele:

- `vCount1` – gültig
- `P_base2` – gültig
- `a_` – ungültig (endet mit Unterstrich)
- `_a` – ungültig (beginnt nicht mit Buchstabe)
- `A` – ungültig (zu kurz)

(b) DFA (Deterministischer endlicher Automat)

Der Automat überprüft den Bezeichner zeichenweise. Er startet im Anfangszustand q_0 und geht je nach Zeichen in andere Zustände über:

- q_0: wartet auf das erste Zeichen, das ein Buchstabe sein muss.
- q_1: hat ein gültiges erstes Zeichen gelesen, aber der Bezeichner ist noch zu kurz.
- q_A: akzeptierender Zustand – der Bezeichner hat mindestens zwei Zeichen und endet auf Buchstabe oder Ziffer.
- q_U: das letzte Zeichen war ein Unterstrich (noch nicht akzeptiert).
- q_{dead}: Fehlerzustand für unzulässige Zeichen oder falschen Aufbau.

Akzeptiert wird nur, wenn das letzte gelesene Zeichen ein Buchstabe oder eine Zahl ist (also Zustand q_A).
Beispiel:

- `vCount1` wird komplett akzeptiert (endet auf Zahl).
- `P_` wird nicht akzeptiert (endet auf Unterstrich).

(c) Reguläre Grammatik

Die Grammatik beschreibt, wie gültige Bezeichner aufgebaut sind:

$$
\begin{array}{ll}
S \to \ell\, T & (\ell \in L) \\
T \to \ell\, R \mid d\, R \mid \_\, U & (\ell \in L,\, d \in D) \\
R \to \varepsilon \mid \ell\, R \mid d\, R \mid \_\, U & (\ell \in L,\, d \in D) \\
U \to \ell\, R \mid d\, R & (\ell \in L,\, d \in D)
\end{array}
$$

- Der Bezeichner beginnt immer mit einem Buchstaben.
- Danach können Buchstaben, Ziffern oder Unterstriche kommen.
- Nach einem Unterstrich muss aber wieder ein Buchstabe oder eine Ziffer folgen.

Ableitung-Beispiel:

$$S \Rightarrow vT \Rightarrow vCR \Rightarrow vCoR \Rightarrow vCouR \Rightarrow vCounR \Rightarrow vCountR \Rightarrow vCount1$$

Das ergibt also den gültigen Bezeichner `vCount1`.

Man sieht also, dass die Regeln gut zusammenpassen und klar beschreiben, wie Bezeichner aussehen dürfen. Manche Fälle sind zwar leicht zu übersehen (z.B. ein Unterstrich am Ende), aber damit sollte eigentlich alles abgedeckt sein

A1.3: Gleitkommazahlen in Programmiersprachen (2P)

Aufbau

In **Python** können Zahlen z.B. so aussehen: `3.0`, `.5`, `10.`, `1.2e3`. In **Java** ist es fast gleich, aber man darf am Ende noch ein Buchstabe f, F, d oder D dran hängen, z.B. `6.02E23f` oder `5d`.

(a) Reguläre Ausdrücke

Python:

```
^[0-9]*\.?[0-9]+([eE][+-]?[0-9]+)?$
```

Java:

```
^[0-9]*\.?[0-9]+([eE][+-]?[0-9]+)?[fFdD]?$
```

(b) DFA

Der Automat liest die Zahl von links nach rechts:

- Startzustand liest Ziffern oder Punkt.
- Danach kommen die Nachkommastellen.
- Wenn e oder E kommt, darf danach + oder - und dann Ziffern folgen.
- In Java kann am Ende noch f, F, d oder D kommen.
- Akzeptiert wird nur, wenn die Zahl mit einer Ziffer oder Typzeichen endet.

(c) Reguläre Grammatik

Python:

$$
\begin{aligned}
&S \to D.D \mid D.DE \mid DE \\
&E \to eD \mid e+D \mid e-D \mid \varepsilon \\
&D \to Z \mid ZD \\
&Z \to 0|1|2|3|4|5|6|7|8|9
\end{aligned}
$$

Java:

$$
\begin{aligned}
&S \to D.DT \mid D.DET \mid DET \\
&T \to f \mid F \mid d \mid D \mid \varepsilon
\end{aligned}
$$

-> Beide Sprachen erlauben Zahlen mit Punkt oder Exponent, aber in Java kann man noch das Typzeichen anhängen. Der rest ist eigentlich gleich. Man muss bissle aufpassen bei den optionalen Teilen, aber vom Prinzip ist es nicht so kompliziert.

A1.4: Mailadressen?

Der gegebene reguläre Ausdruck

$$(a-z)^{+}@(a-z)^{+}(.(a-z)^{+})^{?}$$

ist für echte Mailadressen eher ungeeignet, weil er sehr viele Einschränkungen hat

Warum?:

- Er erlaubt nur kleine Buchstaben, keine Zahlen oder Sonderzeichen.
- Punkte oder Unterstriche im Namen (z.B. `max.mustermann`) sind auch nicht erlaubt.

- Es geht nur eine einfache Domain und eine Endung, also z.B. `abc@xyz.de`, aber nicht `abc@sub.domain.de`.
- Außerdem ist die Schreibweise (a-z) nicht richtig. Eigentlich müsste man jedes Zeichen aufführen oder besser `[a-z]` benutzen.

Darum funktioniert der Ausdruck nur bei sehr einfachen Beispielen, aber nicht bei normalen Mailadressen.

Besserer regulärer Ausdruck

```
^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$
```

Erklärung:

- `[a-zA-Z0-9._%+-]+` erlaubt Buchstaben, Zahlen und manche Sonderzeichen vor dem @.
- `@[a-zA-Z0-9.-]+` erlaubt Domainnamen mit Punkten oder Bindestrichen.
- `\.[a-zA-Z]{2,}` verlangt eine Endung mit min. zwei Buchstaben (z.B. `.de` oder `.com`).

Beispiele:

- `enes.mustermann@mail.de` – gültig
- `ahmad99@uni-bielefeld.de` – gültig
- `@mail.com` – ungültig (Benutzername fehlt)
- `videl@` – ungültig (keine Domain)

A1.5: Der zweitletzte Buchstabe (1P)

Der Automat soll Wörter über dem Alphabet

$$\Sigma = \{1, 2, 3\}$$

akzeptieren, bei denen das zweitletzte Zeichen dasselbe ist wie das zweite.

Idee

Der DFA merkt sich beim Lesen, welches Zeichen an *zweiter Stelle* kam. Dann läuft er weiter durch das Wort und achtet darauf, welches Zeichen jeweils das zweitletzte ist. Am Ende vergleicht er: wenn das zweitletzte Zeichen gleich dem zweiten ist, wird das Wort angenommen.

Man kann sich also vorstellen, dass der Automat quasi immer die letzten zwei Zeichen "im Kopf" hat.

Beispiele

- `2112` → zweites = 1, zweitletztes = 1 → akzeptiert
- `2131` → zweites = 1, zweitletztes = 3 → nicht akzeptiert

Zustände (grob)

- q_0: Startzustand, noch nichts gelesen
- q_1: erstes Zeichen gelesen
- $q_2(1)$, $q_2(2)$, $q_2(3)$: zweites Zeichen gemerkt (1, 2 oder 3)
- danach Zustände, die immer die letzten zwei Zeichen speichern

Akzeptiert wird also nur, wenn am Ende gilt: *zweites Zeichen = zweitletztes Zeichen.*

A1.6: Sprache einer regulären Grammatik (2P)

Gegeben:

$$\begin{aligned} S &\to aA \\ A &\to dB \mid bA \mid cA \\ B &\to aC \mid bC \mid cA \\ C &\to \varepsilon \end{aligned}$$

Analyse & Sprache:

- Jedes Wort startet mit `a` (weil $S \to aA$).
- In A können beliebig viele `b` oder `c` vorkommen.
- Irgendwann muss ein `d` kommen ($A \to dB$).
- Nach jedem `d` kann wieder $B \to cA$ gewählt werden, also nochmal b/c-Folgen.
- Das letzte `d` endet mit $B \to aC$ oder $B \to bC$.

Somit gilt:

$$L = \{\, a(b|c)^*(dc(b|c)^*)^*d(a|b) \,\}$$

Ein passender regulärer Ausdruck ist daher:

$$a(b|c)^*(dc(b|c)^*)^*d(a|b)$$