3 Lexikalische Analyse

3.1 Ziel der lexikalischen Analyse

BEISPIEL: Betrachten wir den folgenden Programmausschnitt:

```
int abs(int x) /* compute absolute value */
{
   if (x > 0) return (x);
   else return (-x);
}
```

Der Übersetzer muss dieses Programm analysieren, um korrekten Code zu erzeugen. Diese Analyse erfolgt wie beim Lesen/Verstehen von Text in zwei Phasen:

1. Die **lexikalische Analyse**: Ziel ist das Erkennen zusammengehöriger „Worte“ wie z.B. `int`, `abs`, `0` etc., aber auch das Überlesen von Kommentaren, Leerzeichen, Zeilenvorschüben etc.

2. Die **syntaktische Analyse**: Erkennung der Programmstruktur (Ausdrücke, `if`-Anweisungen, ...)

Ziel des der lexikalischen Analyse ist also die Zerlegung einer Zeichenfolge (Quellprogramm) in eine Folge von *Lexemen* (lexikalische atomare Einheiten), auch *Token* genannt. Zudem werden die Lexeme in *Symbolklassen* eingeteilt, d.h. in Mengen von Lexeme, die für die syntaktische Analyse als gleichartig anzusehen sind. Ein konkretes Lexem wird bei einer Symbolklasse mit mehreren Elementen als *Attribut* dargestellt.

Klasse	Elemente
Id	`x`, `y`, `abc`, `zaehler1`
Num	`42`, `0`, `99`
Real	`0.3`, `3.14`, `2e-2`
Int	`int`
If	`if`

Klasse	Elemente
LParen	`(`
RParen	`)`
LBrace	`{`
RBrace	`}`

Attribute sind darstellbar als Argumente der Klasse, z.B. `Id "abs"`, `Num 99` etc. Häufig gibt es weitere (implizite) Attribute wie Zeilen- und Spaltennummer des ersten Lexemzeichens (dies ist z.B. wichtig für Fehlermeldungen). Ein Programm für die lexikalische Analyse heißt *Scanner* (*Zerhacker*). Formal ist dies eine Abbildung

$$\text{scan}\colon \Sigma^* \to T^*$$

mit einem Eingabealphabet Σ und einem Tokenklassentyp T.

BEISPIEL: Der Ausdruck scan $\langle$*obiger Programmstring*$\rangle$ ergibt:

```
[Int, Id "abs", LParen, Int, Id "x", RParen, LBrace, If,...]
```

Ein Scanner überliest in der Regel Kommentare, Leerzeichen, ...(nicht aber bei javadoc, lint etc.) und unterscheidet zwischen

- *Schlüsselwörtern* wie `if`, `else`, ...und
- *Bezeichnern* (Klasse `Id`).

Konsequenz ist, dass Schlüsselworte nicht als Bezeichner verwendbar sind, beispielsweise ist eine Anweisung wie

```
int int; int = 3;
```

meist unzulässig.

3.2 Reguläre Ausdrücke

Beobachtung für die Implementierung: Die Struktur der Tokenklassen ist regulär, damit kann ein Scanner durch reguläre Ausdrücke und somit durch endliche Automaten beschrieben werden.

DEFINITION (*regulärer Ausdruck*): Sei Σ ein Alphabet. Die Menge $\mathrm{RA}(\Sigma)$ aller regulären Ausdrücke über Σ ist die kleinste Menge mit folgenden Eigenschaften:

- $\Sigma \cup \{\varnothing, \varepsilon\} \subseteq \mathrm{RA}(\Sigma)$ ($\varnothing$: leere Menge; ε: leere Zeichenfolge)
- $\alpha, \beta \in \mathrm{RA}(\Sigma) \Longrightarrow \alpha|\beta \in \mathrm{RA}(\Sigma), \alpha \cdot \beta \in \mathrm{RA}(\Sigma), \alpha^* \in \mathrm{RA}(\Sigma)$

Jeder reguläre Ausdruck beschreibt eine Menge (Sprache) von Zeichenkettten. Daher ist die Semantik der regulären Ausdrücke durch eine Funktion $[\![\cdot]\!]\colon \mathrm{RA}(\Sigma) \to 2^{\Sigma^*}$ definiert:

$$
\begin{aligned}
[\![\varnothing]\!] &= \{\} \\
[\![\varepsilon]\!] &= \{\varepsilon\} \\
[\![a]\!] &= \{a\} \\
[\![\alpha|\beta]\!] &= [\![\alpha]\!] \cup [\![\beta]\!] \\
[\![\alpha \cdot \beta]\!] &= \{\, A \cdot B \mid A \in [\![\alpha]\!], B \in [\![\beta]\!] \} \\
[\![\alpha^*]\!] &= \{\, A_1 \cdots A_n \mid A_1, \ldots, A_n \in [\![\alpha]\!], n \geq 0 \}
\end{aligned}
$$

Weitere Abkürzungen, die in vielen Werkzeugen zur Verarbeitung regulärer Ausdrücke verwendet werden, sind z.B.

- $[a-z]$ ist die Menge aller Zeichen zwischen a und z, also $a|b|c|\cdots|z$
- α^+ bedeutet $\alpha \cdot \alpha^*$ (Wiederholung, mindestens einmal)

BEISPIEL: In der konkreten Notation wird der Punkt "$\cdot$" bei der Konkatenation oft weggelassen:

$$\begin{aligned} \texttt{Id} &= [a-z]([a-z]|[0-9])^* \\ \texttt{Num} &= [0-9]^+ \\ \texttt{If} &= if \end{aligned}$$

3.3 Implementierung

Die Übersetzung eines regulären Ausdrucks in ein Scanner-Programm kann in mehreren Schritten erfolgen.

3.3.1 Übersetzung in einen endlichen Automaten

DEFINITION: Ein *nichtdeterministischer endlicher Automat (NEA, NFA)* ist ein Tupel $A = (Q, \Sigma, \delta, q_0, F)$ mit Zuständen Q, Alphabet Σ, Startzustand $q_0 \in Q$, Endzuständen $F \subseteq Q$ und einer Zustandübergangsfunktion

$$\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$$

BEISPIEL: Ein Automat zur Erkennung von `Id` und `If` (siehe Vorlesung). Hierbei können die folgenden Unklarheiten entstehen:

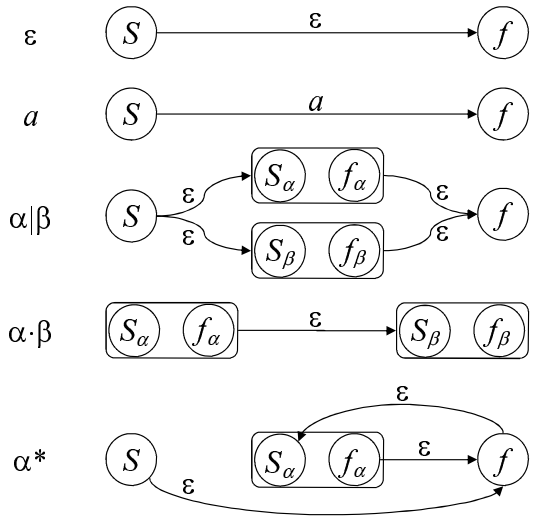
1. Ist die Zeichenfolge "`if`" ein `Id` oder ein `If`?
2. Ist die Zeichenfolge "`ifxy`" ein Lexem (`Id "ifxy"`) oder zwei (`If`, `Id "xy"`)?

Übliche Festlegung:

zu 2. Benutze das *principle of longest match*: Der längste Präfix, der zu einer Symbolklasse gehört, ist das nächste Token. Damit ist "`ifxy`" ein Token, dagegen sind "`if(`" sind zwei Token.

zu 1. Die erste Symbolklasse, zu der ein „longest-match"-Token gehört, ist die Klasse dieses Tokens. Damit ist die Reihenfolge der Symbolklassen relevant, üblicherweise werden die Schlüsselwörter vor den Identifiern genannt.

Zur genauen Beschreibung der Übersetzung eines regulären Ausdrucks in einen NFA nehmen wir an, dass jeder NFA hat einen Start- und einen Endzustand hat (dies ist immer möglich mittels ε-Transitionen). Dann können wir reguläre Ausdrücke folgendermaßen in NFAs übersetzen:



Bei dieser Übersetzung kann man redundante Zustände gleich verschmelzen.

3.3.2 Übersetzung eines NFA in einen DFA

Problem der NFA-Übersetzung: Ein Rechner kann nicht „raten“, daher ist ein NFA nicht zur Implementierung geeignet.
Lösung: Konvertiere den NFA in einen DFA (wie NFA, aber mit eindeutiger (partieller) Übergangsfunktion $\delta\colon Q \times \Sigma \dashrightarrow Q$). Es fehlen jedoch die ε-Übergänge, die ein NFA machen kann. Daher repräsentieren wir eine Menge von NFA-Zuständen als einen DFA-Zustand.

Definition: Sei ein NFA $(Q, \Sigma, \delta, q_0, F)$ gegeben und $T \subseteq Q$. Der ε-Abschluß $\hat{\varepsilon}(T)$ von T ist die kleinste Menge mit

- $T \subseteq \hat{\varepsilon}(T)$ und
- $q \in \hat{\varepsilon}(T) \Rightarrow \delta(q, \varepsilon) \subseteq \hat{\varepsilon}(T)$.

Algorithmus zum Berechnen des ε-Abschlusses:

```
S := T;
repeat
    S' := S;
    S := S ∪ ⋃_{q∈S}, δ(q,ε)
until S == S'
return S;
```

Der Algorithmus terminiert, da Q endlich ist und in jeder Iteration immer nur Zustände hinzugefügt werden.

Damit kann man einen NFA deterministisch simulieren:

```
T := ε̂({q0});
while not final(T)
do
    read c;
    T := ε̂(⋃_{q∈T} δ(q,c))
done
```

Hierbei ist `final(T) = (`$T \cap F \neq \varnothing$`)` (Endzustände).

Die Berechnung des ε-Abschlusses ist jedoch aufwendig, besser ist eine Vorberechnung bei der Konstruktion des Übersetzers durch die *Potenzmengenkonstruktion*: Definiere zu einem NFA $(Q, \Sigma, \delta, q_0, F)$ einen DFA $(Q', \Sigma, \delta', q_0', F')$ durch

- $Q' = 2^Q$ (kann später auf erreichbare Zustände reduziert werden)
- $F' = \{T \subseteq Q \mid T \cap F \neq \varnothing\}$
- $q_0' = \hat{\varepsilon}(\{q_0\})$
- $\delta' \colon Q' \times \Sigma \to Q'$ sei definiert durch $\delta'(T, a) = \hat{\varepsilon}\left(\bigcup_{q \in T} \delta(q, a)\right)$

Der konstruierte Automat enthält zu viele Zustände („Zustandsexplosion“). Diese Anzahl kann reduziert werden:

- Erzeuge nur nur die von q_0' erreichbaren Zustände, d.h. starte mit q_0' und berechne den iterativ alle weiteren erreichbaren Zustande.
- Verschmelze „äquivalente“ Zustände, d.h. Zustände, von denen aus nur gleiche Zeichenfolgen erkannt werden können („Automatenminimierung“).

3.3.3 Praktische Aspekte der Scannerimplementierung

- Schreibe an Endzustände erkannte Symbolklasse zur Scannerausgabe (bei mehreren Tokenklassen: die textuell erste, siehe oben)!
- Umsetzung des „longest match“: Merke letzten erfolgreichen Endzustand und Position; bei Nichtendzuständen ohne weiteren Übergang setze auf den letzten (gemerkten) Endzustand zurück.

- Bei vielen Schlüsselwörtern entstehen viele Zustände – Reduktion: Erkenne Schlüsselwörter genauso wie Bezeichner, falls der Scanner dann "`Id x`" erkennt, so prüfe, ob `x` ein Schlüsselwort ist: wenn ja, dann gib das entsprechende Token zurück, andernfalls "`Id x`". Dieses Vorgehen ist vor allem empfehlenswert bei handgeschriebenen Scannern.

- Implementierung des DFA als Programm: Entweder durch eine Tabelle (δ') mit einen „Interpreter" dafür oder durch Übersetzung in `case`-Ausdrücke.

- Die **Scannerimplementierung** ist damit voll automatisierbar. Daher existieren vollständige **Scanner-Generatoren** (z.B. lex), die reguläre Ausdrücke und zugehörige Symbolklassen in ein Scanner-Programm übersetzen.