8 Syntax von Programmiersprachen und Syntaxanalyse

Mit Hilfe des Pumping Lemmas wurde im vorigen Abschnitt gezeigt, dass schon einfachste Klammerstrukturen mit n öffnenden Klammern (repräsentiert durch a) gefolgt von n schließenden Klammern (repräsentiert durch b) für beliebige $n \in \mathbb{N}$ nicht von endlichen Automaten erkannt werden können. Das ist auch anschaulich klar. Denn beim Lesen von links nach rechts kann sich ein endlicher Automat mit seiner beschränkten Zahl von Zuständen nur beschränkt viele öffnende Klammern merken, während das Eingabewort beliebig viele enthalten kann, so dass später kein Vergleich mehr mit der Zahl der schließenden Klammern möglich ist. Da Klammerstrukturen in praktisch allen Programmiersprachen vielfältig vorkommen, für die dann eine analoge Beweisführung möglich ist wie für die einfachen Klammerstrukturen, stellt sich heraus, dass Programme von Programmiersprachen in der Regel nicht durch endliche Automaten erkannt werden können.

Wer ein Programm in einer Programmiersprache X schreiben möchte, wählt sich häufig einen Texteditor Y aus und erstellt mit dessen Hilfe eine bestimmte Zeichenkette, die dann als Eingabe für den Compiler von X dient. Dies ist in Abbildung 5 skizziert.

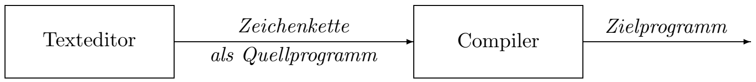

Abbildung 5: Erstellung eines Programms

Man kann nicht erwarten, dass jeder Text, der mit dem Editor Y entstehen kann, bereits ein Programm in X ist. Denn mit einem Texteditor lassen sich in der Regel beliebige Zeichenketten aufbauen, während ein Programm eine bestimmte Form haben muss. Zeichenketten jedoch, die keine Programme sind, wird der Compiler als unübersetzbar zurückweisen. Woher weiß aber eine Programmiererin oder ein Programmierer, wie die Zeichenkette aussehen muss, um ein Programm zu sein? Wie findet der Compiler heraus, ob irgendeine Zeichenkette ein übersetzbares Programm darstellt?

Die Form von Programmen einer Programmiersprache wird durch ihre Syntax festgelegt. Die Syntaxdefinition macht meist äußerst restriktive Vorschriften über die Anordnung und Plazierung von Zeichen, damit eine Zeichenkette ein syntaktisch richtiges Programm ist. Eine Person, die ein Programm mit Hilfe eines Texteditors erstellen will, sollte die Syntax recht gut kennen, weil andernfalls sicherlich häufig Syntaxfehler auftreten. Der Compiler dagegen übersetzt die eingegebene Zeichenkette in ein Zielprogamm, falls die Eingabe ein syntaktisch korrekt gebildetes Programm ist. Um das festzustellen, besitzen Compiler eine Syntaxanalyse-Komponente, die diese Aufgabe übernimmt.

Bei der Syntaxanalyse wird für eingegebene Zeichenketten untersucht, ob sie Programme

sind oder nicht. Im positiven Fall wird außerdem der syntaktische Aufbau ermittelt, weil diese Information bei der weiteren Übersetzung maßgeblich genutzt wird. Im negativen Fall werden meist noch Hinweise auf Syntaxfehler gegeben. Die Trennung in "richtige" und "falsche" Zeichenketten ist in der Regel das entscheidende algorithmische Problem, weil bei der Lösung die zusätzlichen Informationen ohne allzu große Mühe nebenbei gewonnen werden können.

Die syntaktisch richtigen Programme einer Programmiersprache bilden als Menge von Zeichenketten eine "formale Sprache". Solche Zeichenkettenmengen sind Gegenstand der Untersuchung in der Theorie formaler Sprachen, die Konzepte für die syntaktische Definition formaler Sprachen bereitstellt und Methoden liefert, um die Eigenschaften formaler Sprachen analysieren zu können. Zu den wichtigsten Anwendungsfeldern der Theorie formaler Sprachen gehören die Syntaxdefinition von Programmiersprachen und die Syntaxanalyse. Die Schlüsselfrage der Syntaxanalyse, ob eine Zeichenkette ein Programm ist oder nicht, wird auch *Wortproblem* genannt. Betrachtet man die Menge aller richtigen Programme als formale Sprache, dann besteht das Problem darin, ob eine Zeichenkette in der Sprache liegt oder nicht. Da die Lösung des Wortproblems eine zentrale Rolle bei der Implementierung von Programmiersprachen spielt, aber keineswegs immer auf der Hand liegt, zieht sich die Behandlung des Wortproblems wie ein roter Faden durch die Theorie formaler Sprachen.

Aber erst einmal zurück zur Syntaxdefinition. Eine grundlegende Weise, formale Sprachen, einschließlich Programmiersprachen, zu spezifizieren, besteht darin, die übliche Art der Begriffsbildung (unter *dem und dem* versteht man *das und das*) in formalisierter Form zu nutzen. Einem zu definierenden, also noch undefinierten, syntaktischen Konstrukt wird ein definierender Ausdruck zugeordnet. Beides zusammen bildet eine Syntaxregel, das (noch) Undefinierte wird linke, das Definierende rechte Regelseite genannt. Der definierende Ausdruck der rechten Seite ist eine Zeichenkette, die rekursiv auch wieder undefinierte Konstrukte enthalten darf. Ist das noch Undefinierte der linken Seite durch ein einziges Zeichen dargestellt, spricht man von einer kontextfreien Regel. Die Syntaxdefinition einer Programmiersprache besteht in einem ersten Anlauf meist in der Angabe von kontextfreien Regeln, die dann später um Syntaxteile ergänzt werden, die sich nicht durch kontextfreie Regeln ausdrücken lassen.

Der kontextfreie Anteil der Syntax von Programmiersprachen wird häufig in der sogenannten Backus-Naur-Form geschrieben, wobei die kontextfreien Regeln als linke Seiten nichtterminale Zeichen, die zu definierende syntaktische Konstrukte der Sprache benennen, und als rechte Seiten Zeichenketten aus terminalen und nichtterminalen Zeichen besitzen. Die rechten Seiten zur selben linken Seite werden als Alternativen nebeneinandergestellt, durch einen senkrechten Strich voneinander getrennt. Linke und rechte Seiten werden durch das Trennzeichen "::=" auseinandergehalten. Nichtterminale Zeichen sind in spitze Klammern eingeschlossen.

Für die Beschreibung der Form von Programmen – oder wie man auch sagt: ihrer Syntax – werden also andere Mittel gebraucht. Sehr häufig werden dafür kontextfreie Grammatiken verwendet, die in ähnlicher Form auch für die grammatikalische Beschreibung natürlicher

Sprachen eingesetzt werden. In diesem Abschnitt werden solche Grammatiken motiviert und informell eingeführt. Die formale Behandlung folgt dann in den folgenden Abschnitten.

Um das Prinzip zu illustrieren, sollen Boolesche Ausdrücke einfacher Art beschrieben werden. Ein solcher Ausdruck kann eine der Booleschen Konstanten $true$ oder $false$ sein, eine Boolesche Variable, ein negierter Boolescher Ausdruck oder die Komposition zweier Boolescher Ausdrücke durch einen Booleschen Operator. In den beiden letzten Fällen werden jeweils Klammern verwendet, damit Anfang und Ende der Ausdrücke eindeutig festgelegt sind. Neben den zu definierenden Ausdrücken sind auch die Variablen ein syntaktisches Konstrukt, das definiert werden muss. Der Einfachheit halber geschieht das durch jeweils ein $''b''$ gefolgt von einer Ziffernfolge. Desweiteren sind Boolesche Operatoren und Ziffernfolge sowie Ziffern als syntaktische Konstrukte eingeführt und definiert. Die folgenden Syntaxregeln reflektieren die verbale Beschreibung:

$$
\begin{aligned}
&< boolexp >::= true \mid false \mid < var > \mid \\
&(\neg < boolexpr >) \mid (< boolexpr >< boolop >< boolexpr >) \\
&< boolop >::= \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \\
&< var >::= b < cipherseq > \\
&< cipherseq >::=< cipher > \mid < cipher >< cipherseq > \\
&< cipher >::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Die Regeln können zum Aufbau syntaktisch korrekter Boolescher Ausdrücke benutzt werden, indem mit dem nichtterminalen Zeichen $< boolexpr >$ begonnen und dann nach und nach in den aktuellen Zeichenketten je ein nichtterminales Zeichen durch eine zugehörige rechte Regelseite ersetzt wird, bis keine nichtterminalen Zeichen mehr vorkommen. Ein Beispiel dafür ist:

$$
\begin{aligned}
< boolexp > &\rightarrow (< boolexpr >< boloop >< boolexpr >) \\
&\rightarrow (< boolexpr > \Leftrightarrow < boolexpr >) \\
&\rightarrow (< boolexpr > \Leftrightarrow true) \\
&\rightarrow ((< boolexpr >< boloop >< boolexpr >) \ \Leftrightarrow true) \\
&\rightarrow ((< boolexpr > \vee < boolexpr >) \Leftrightarrow true) \\
&\rightarrow ((< boolexpr > \vee (\neg < boolexpr >)) \ \Leftrightarrow true) \\
&\rightarrow ((< var > \vee (\neg < boolexpr >)) \ \Leftrightarrow true) \\
&\rightarrow ((< var > \vee (\neg < var >)) \Leftrightarrow true) \\
&\rightarrow ((b < cipherseq > \vee (\neg < var >)) \ \Leftrightarrow true) \\
&\rightarrow ((b < cipherseq > \vee (\neg\ b < cipherseq >)) \ \Leftrightarrow true) \\
&\rightarrow ((b < cipher > \vee (\neg\ b < cipherseq >)) \ \Leftrightarrow true) \\
&\rightarrow ((b < cipher > \vee (\neg\ b < cipher >)) \ \Leftrightarrow true) \\
&\rightarrow ((b < cipher > \vee (\neg\ b\ 0)) \ \Leftrightarrow true) \\
&\rightarrow ((b\ 0 \vee \ (\neg\ b\ 0)) \ \Leftrightarrow true)
\end{aligned}
$$

Wie dieses Gesetz vom ausgeschlossenen Dritten lassen sich alle Booleschen Ausdrücke mit Hilfe der Regeln herleiten.