9 Theoretische Informatik und Compilerbau

Theoretische Informatik und Mathematik schaffen die Basis für viele der technischen Entwicklungen, die wir in diesem Buch besprechen. Die *boolesche Algebra* (S. 419 ff.) legt die theoretischen Grundlagen für den Bau digitaler Schaltungen, die *Theorie formaler Sprachen* zeigt auf, wie die Syntax von Programmiersprachen aufgebaut werden sollte, damit Programme einfach und effizient in lauffähige Programme übersetzt werden können, und die *Theorie der Berechenbarkeit* zeigt genau die Grenzen des Berechenbaren auf. Sie zieht eine klare Linie zwischen dem was man prinzipiell programmieren kann und dem was mit Sicherheit nicht von einem Rechner gelöst werden kann.

In diesem Kapitel wollen wir einen kurzen Ausflug in die theoretische Informatik unternehmen. Wir werden sehen, dass diese Theorie nicht trocken ist, sondern unmittelbare praktische Anwendungen hat. Die *Automatentheorie* zeigt, wie man effizient die Wörter einer Programmiersprache festlegen und erkennen kann, die *Theorie der kontextfreien Sprachen* zeigt, wie man die Grammatik einer Programmiersprache definieren sollte, und wie man Übersetzer und Compiler dafür bauen kann. Letzterem haben wir ein eigenes Unterkapitel *Compilerbau* gewidmet. Ein fehlerfreies Programm ist aber noch lange nicht korrekt. Wünschenswert wäre es, wenn man feststellen könnte, ob jede Schleife auch terminiert. Dass diese und ähnliche semantischen Eigenschaften nicht automatisch geprüft werden können, ist eine der Konsequenzen der *Berechenbarkeitstheorie*. Mit dieser kann man zeigen, dass, wenn man einmal von Geschwindigkeit und Speicherplatz absieht, alle Rechner in ihren mathematischen Fähigkeiten identisch sind und damit das gleiche können bzw. nicht können. Schließlich hilft uns die *Komplexitätstheorie,* Aussagen über den Aufwand zu machen, den man zur Lösung wichtiger Probleme treiben muss.

9.1 Analyse von Programmtexten

Programme sind zuerst einmal Texte, also Folgen von Zeichen aus einem gewissen Alphabet. Die ersten Programmiersprachen erlaubten nur Großbuchstaben, Klammern und Ziffern. In Pascal und C sind auch Kleinbuchstaben und Sonderzeichen, wie _ , +, (,), >, : , möglich und in Java sind sogar Unicode-Zeichen, insbesondere auch ä, ö und ü erlaubt. Die Menge aller Zeichen, die in einem Programm einer gewissen Programmiersprache vorkommen dürfen,

nennt man ihr *Alphabet*. Aus diesem Alphabet definiert man zunächst einmal die *Wörter*, aus denen die Sprache aufgebaut werden soll. Im Falle von Java oder Pascal sind dies u.a. *Schlüsselwörter* (**while**, **do**, **for**, **if**, **else**, ...), *Sonderzeichen* (+, **-**, *, <=, >= , ...), benutzerdefinierte *Bezeichner* (`testFunktion, betrag, _anfangsWert, x37, r2d2`) und *Konstanten*. Unter letzteren unterscheidet man noch Integer-Konstanten (`42, 386, 2004`) von Gleitkommazahlen (`3.14, 6.025e23, .5`).

Ein Compiler für eine Programmiersprache muss als erstes prüfen, ob eine vorgelegte Datei ein syntaktisch korrektes Programm enthält. Ihm liegt der Quelltext als String, also als eine Folge von Zeichen vor. Die Analyse des Textes zerfällt in zwei Phasen – die *lexikalische Analyse* und die *syntaktische Analyse*. Dies entspricht in etwa auch unserem Vorgehen bei der Analyse eines fremdsprachlichen Satzes: In der ersten Phase erkennen wir die Wörter, aus denen der Satz besteht – vielleicht schlagen wir sie in einem Lexikon nach – und in der zweiten Phase untersuchen wir, ob die Wörter zu einem grammatikalisch korrekten Satz zusammengefügt sind.

9.1.1 Lexikalische Analyse

Die erste Phase eines Compilers nennt man *lexikalische Analyse*. Dabei wird der vorgebliche Programmtext in Wörter zerlegt. Alle Trennzeichen (Leerzeichen, Tab, newLine) und alle Kommentare werden entfernt. Aus einem einfachen PASCAL-Programm, wie

```
PROGRAM ggT;
BEGIN
    x := 54;
    y := 30;
    WHILE not x = y DO
     IF x > y THEN x := x-y ELSE y:= y-x;
    writeln('Das Ergebnis ist: ',x)
END .
```

wird dabei eine Folge von *Token*. Dieses englische Wort kann man mit *Gutschein* übersetzen. Für jede ganze Zahl erhält man z.B. ein Token ***num***, für jeden Bezeichner ein Token ***id***, für jeden String ein Token ***str***. Gleichartige Token stehen für gleichartige Wörter. Andere Token, die in dem Beispielprogramm vorkommen, sind ***eq*** (=) , ***gt*** (>), ***minus*** (**-**), ***assignOp***(**:=**), ***klAuf*** (**(**), ***klZu***(**)**), ***komma*** (**,**), ***semi*** (**;**), ***punkt*** (**.**). Jedes Schlüsselwort bildet ein Token für sich.

Nach dieser Zerlegung ist aus dem Programm eine Folge von Token geworden. Damit ist die erste Phase, die lexikalische Analyse, abgeschlossen. Im Beispiel hätten wir:

program id semi begin id assignOp num semi id assignOp num semi
while not id eq id do if id gt id then id assignOp id minus id else
id assignOp id minus id semi id klAuf str komma id klZu end punkt

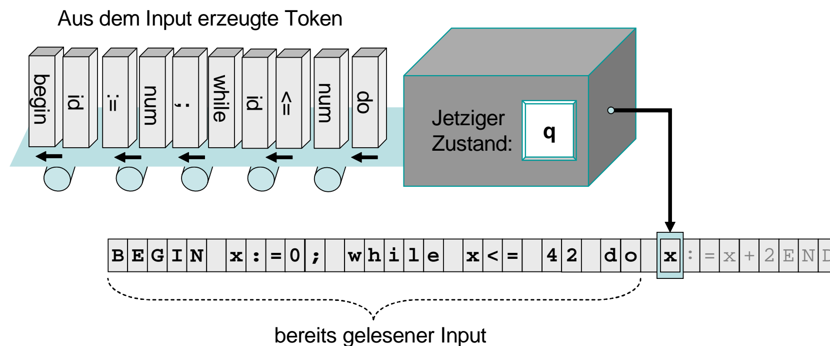

Abb. 9.1: *Endlicher Automat als Scanner*

Diese erste Phase ist nicht schwer, aber auch nicht trivial. Es muss u.a. entschieden werden, ob `BeGin` das gleiche ist, wie `BEGIN`, ob `beginnen` ein ***id*** ist, oder das Schlüsselwort ***begin*** gefolgt von dem Bezeichner `nen`. Schließlich muss erkannt werden, ob `00.23e-001, .523e, 314.e-2` das Token ***num*** repräsentieren, oder nicht. Programme für die lexikalische Analyse heißen *Scanner*. Heutzutage kann man solche Scanner aus einer Beschreibung der Token durch sogenannte *reguläre Ausdrücke* automatisch erzeugen. Der bekannteste frei erhältlicher Scannergenerator heißt `flex`. Die lexikalische Analyse ist das Thema des ersten Unterkapitels.

9.1.2 Syntaxanalyse

Wir nehmen an, dass unser Programm die lexikalische Analyse gut überstanden hat. Dann folgt als nächstes die sogenannte *Syntaxanalyse*. Hier wird geprüft, ob die gefundenen Wörter in der vorliegenden Reihenfolge ein grammatikalisch korrektes Programm bilden.

Ausdrücke wie *Wörter*, *Sprache*, *Grammatik*, etc. erinnern nicht zufällig an natürliche Sprachen. Auch eine Analyse eines fremdsprachigen Satzes erfordert zunächst seine Zerlegung in Wörter und deren Klassifikation. Dabei könnte man vielleicht Token wie ***verb***, ***nomen***, ***artikel***, ***adjektiv***, ***komma***, ***punkt*** benutzen. Aus einem Satz wie beispielsweise

```
Die lila Kuh legt ein Schokoladenei.
```

hätte die lexikalische Analyse die Tokenfolge

artikel adjektiv nomen verb artikel nomen punkt

abgeliefert. In der syntaktischen Analyse muss jetzt anhand der Regeln der Sprache überprüft werden, ob diese Folge von Token zulässig ist.

Hier würde man Bildungsregeln der deutschen Sprache heranziehen, wie (stark vereinfacht):

Satz	:: *Subjekt Prädikat Objekt*
Subjekt	:: ***artikel nomen*** | ***artikel adjektiv nomen***
Prädikat	:: ***verb*** | ***hilfsverb***
Objekt	:: ***artikel nomen***

Ganz analog sehen die Bildungsregeln einer Programmiersprache aus. Im Beispiel von Pascal hat man u.a.:

Programm	:: *Kopf Deklarationen AnweisungsTeil* ***punkt***
Kopf	:: ***program id semi***
Anweisungsteil	:: ***begin*** *Anweisungen* ***end***
Anweisungen	:: *Anweisung* | *Anweisungen Anweisung*

In der Syntaxanalyse überprüft man, ob eine Folge von Token diesen Regeln entsprechen. Diesen Überprüfungsprozess nennt man auch *Parsen* (engl.: *to parse* = zerlegen).

Wie man Regeln für eine Programmiersprache sinnvollerweise festlegt, und wie man einen Parser dafür schreiben und sogar automatisch erzeugen kann, davon handelt der zweite Teil dieses Kapitels.

9.2 Reguläre Sprachen

Zuerst erkennen wir eine Gemeinsamkeit von lexikalischer und syntaktischer Analyse. Erstere betrachtet Wörter als Folge von Zeichen und fragt, ob ein Wort zu einer Klasse von Wörtern (***num***, ***float***, ***id***) gehört, letztere betrachtet Wörter als Folge von Token und fragt ebenfalls, ob das zusammengesetzte Wort zur Klasse der nach gewissen Regeln korrekt aufgebauten Programme gehört. Abstrakt definieren wir einfach:

Definition (Alphabet): *Ein Alphabet ist eine endliche Menge von Zeichen.*

Alphabete bezeichnet man gern mit großen griechischen Buchstaben wie Σ oder Γ und Zeichen mit a,b,c, etc. Aus den Zeichen eines Alphabets können wir Wörter bilden, daher folgt sogleich die nächste Definition:

Definition (Wort): *Ein Wort über einem Alphabet Σ ist eine endliche Folge von Zeichen aus Σ. Die Menge aller Wörter über Σ bezeichnet man mit Σ^*.*

Wörter bezeichnen wir meist mit den Buchstaben u, v, w. Eine besondere Rolle spielt das *leere Wort*, es wird mit ε bezeichnet. Wörter über Σ sind nichts anderes als *Strings*, deren Zeichen aus Σ stammen.

Als *Konkatenation* bezeichnet man die Zusammenfügung zweier Wörter u und v zu einem neuen Wort, das man als uv bezeichnet. Für jedes Wort u gilt offensichtlich: $\varepsilon u = u\varepsilon = u$.

Die *Länge* eines Wortes w ist die Anzahl der Zeichen, aus denen w besteht. Man schreibt $|w|$ für die Länge von w. Offensichtlich gilt: $|\varepsilon|=0$ und $|uw| = |u|+|w|$. Ein Zeichen kann man auch

als Wort der Länge 1 auffassen. Daher benutzen wir die Notation *aw* auch für das Wort, das aus *w* entsteht, wenn man das Zeichen *a* davorsetzt, analog versteht man *wa*.

Beispiele: Sei Σ die Menge aller Ziffern, und Γ die Menge aller Großbuchstaben, dann sind 42, 2004, 0, ε Elemente von Σ^* und ε, C, ML, LISP sind Elemente von Γ^*. Weiter sind R2D2, MX5 und X86 Elemente von $(\Sigma \cup \Gamma)^*$, nicht aber von $\Sigma^* \cup \Gamma^*$.

Definition (Sprache): *Eine* Sprache *über dem Alphabet* Σ *ist eine Menge von Wörtern über* Σ.

Eine Sprache über Σ ist also einfach eine beliebige Teilmenge von Σ^*. Insbesondere ist auch die leere Menge $\{ \}$ eine Sprache, ebenso wie Σ^*. Für Sprachen benutzt man gerne den Buchstaben *L*, weil dieser an das englische Wort *language* erinnert.

Die folgenden Beispiele sind relevant und motivierend für die Entwicklung der Theorie:

Beispiel 1: Die Sprache der Pascal-Bezeichner. Als Alphabet hat man die Ziffern, die Buchstaben (ohne Umlaute) und den Unterstrich, also $\Sigma = \{_, 0, ..., 9, a, ..., z, A, ..., Z\}$. Die Sprache aller gültigen Pascalbezeichner ist $L = \{au \mid u \in \Sigma^*, a \notin \{0, \ldots, 9\}\}$, also alle nichtleeren Wörter, die nicht mit einer Ziffer beginnen.

Beispiel 2: Die Sprache aller dezimalen Konstanten in Assembler. Dezimale Konstanten sind alle Wörter, die optional mit einem Vorzeichen beginnen, gefolgt von einer Folge von Ziffern. Sie dürfen nicht mit 0 beginnen außer wenn keine weitere Ziffer folgt. (In Assembler wird 012 als Hex-Zahl interpretiert.) Als Alphabet wählen wir $\Gamma = Digit \cup Sign$ mit $Digit = \{0, 1, \ldots, 9\}$ und $Sign = \{-, +\}$. Wir definieren zuerst eine Hilfssprache $L_{nat} = \{au|a \in Digit - \{0\}, u \in Digit^*\} \cup \{0\}$ und dann schließlich

$$L_{DAss} = L_{nat} \cup \{au|a \in Sign, u \in L_{nat}\}.$$

Mit einem kleinen Trick hätten wir die Definition kompakter gestalten können: Mit Hilfe des leeren Wortes definieren wir uns eine Sprache $OptSign = \{+, -, \varepsilon\}$. Dann haben wir $L_{DAss} = \{uv|u \in OptSign, v \in L_{nat}\}$.

Offensichtlich lässt die bisher verfügbare mathematische Notation noch Wünsche offen. Um z.B. die Sprache aller *float*-Zahlen auf die obige Weise in Java auszudrücken, müsste man sich schon anstrengen. Wir erinnern, dass u.a. die folgenden Zahlen gültige float sind: `-3.14`, `2.7f`, `2.7`, `.3`, `1e-9F`, aber auch `00.5E000F`.

9.2.1 Reguläre Ausdrücke

Um komplizierte Sprachen aus einfacheren aufbauen zu können, definieren wir uns die Operationen, mit denen wir aus gegebenen Sprachen neue konstruieren können:

Seien *L* und *M* Sprachen über dem gemeinsamen Alphabet Σ. Wir definieren

$L \cdot M = \{ uv \mid u \in L, v \in M\}$, das *Produkt* von *L* und *M,*

$L^0 = \{\varepsilon\}$, $L^{n+1} = L \cdot L^n$, die *Potenzen* von *L,*

$L^* = \bigcup \{ L^n \mid n \in Nat \}$, der *Kleene-Stern* von *L.*