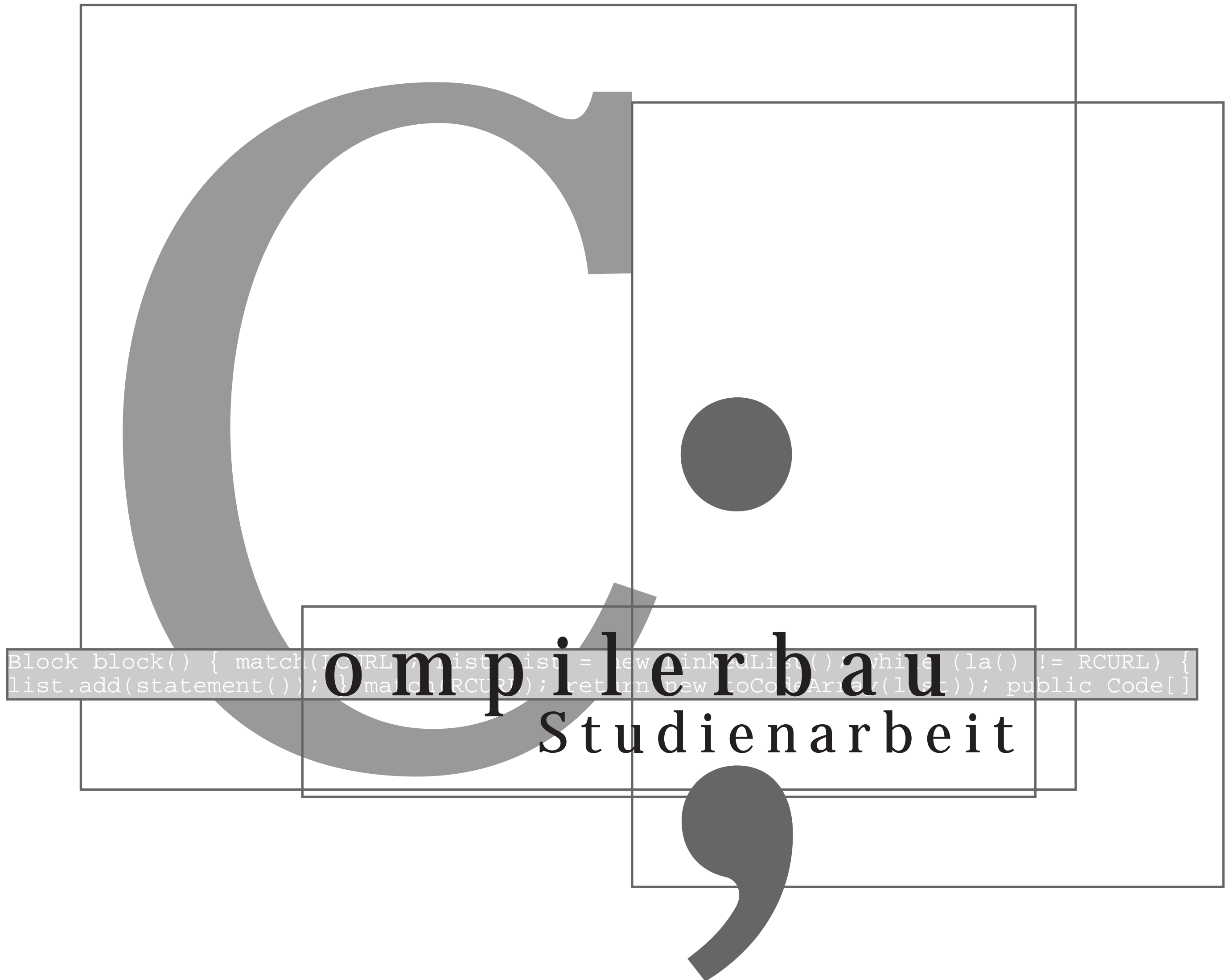
Oliver Skawronek



vorgelegt im:	Oktober 2010
Matrikelnummer:	G080213PI
Studiengang:	Praktische Informatik
Kurs:	PI08
Betreuer:	Prof. Dr.-Ing. Wolfram Rauschenbach

Inhaltsverzeichnis

Abbildungsverzeichnis III

Abkürzungsverzeichnis IV

1 Einleitung und Aufgabenstellung 1

1.1 Aufgabenstellung 2

2 Übersicht Kompilierung 3

2.1 Lexikalische Analyse 3

2.2 Syntaxanalyse 4

2.3 Semantische Analyse 5

2.4 Synthesephase 7

3 Grundlagen Wort und Sprache 10

3.1 Wort 10

3.2 Sprache 11

4 Lexikalische Analyse 12

4.1 Reguläre Ausdrücke und reguläre Sprachen 13

4.2 Endliche Automaten 16

4.3 Konstruktion endlicher Automaten aus regulären Ausdrücken 19

4.4 Simulation endlicher Automaten 22

4.5 Entwurf eines Scanners 25

5 Syntaxanalyse 27

5.1 Kontextfreie Grammatiken 27

5.2 Ableitungsbäume 30

5.3 Links- und Rechtsableitung 32

5.4 LL(k)-Grammatiken 32

5.5 Elimination von Links-Rekursionen 34
5.6 Links-Faktorisierung 35
5.7 Entwurf eines Recursive-Descent-Parsers 36

6 Semantische Analyse und Synthesephase 39

6.1 Abstrakte Syntaxbäume 40
6.2 Optimierung des abstrakten Syntaxbaums 41
6.3 Code-Generierung 42
6.4 Peephole-Optimization 46

7 Zusammenfassung und Ausblick 47

Quellen- und Literaturverzeichnis 48

Abbildungsverzeichnis

1 Wesentliche Bestandteile eines Compilers 4

2 Ableitungsbaum für `a = b + 10;` 5

3 Übergangsfunktion für den DEA aus Abbildung 4 18

4 Ein DEA, der die Vergleichsoperatoren `==`, `<=` und `>=` erkennt 18

5 NEAs aus den regulären Ausdrücken a) ϵ, b) $\emptyset$ und c) $a \in \Sigma$ 19

6 Konstruktion von M für $r|s$ aus M_1 und M_2 20

7 Konstruktion von M für rs aus M_1 und M_2 20

8 Konstruktion von M für r^* aus M_1 21

9 Dekomposition von $r = a((b|c)|\epsilon)$ 21

10 NEAs für a, b, c, ϵ (links) und $r_3 = b|c$ (rechts) 22

11 NEAs für $r_2 = (b|c)|\epsilon)$ 22

12 NEAs für $r_1 = a((b|c)|\epsilon)$ 22

13 Entwurf eines Scanners aus mehreren Teil-NEAs 26

14 Ableitungsbaum für $(01 + 1)$ 31

15 Ableitungsbaum für $S \Rightarrow^* ab$ 31

16 Ableitungsbäume für $S \Rightarrow AA \Rightarrow AAA \Rightarrow^3 aaa$ 32

17 Konstruierter AST für $a - 4 + c$ 41

18 Unoptimierter AST für $a + (1 + 2) - 3$ 42

19 Optimierter AST für $a + (1 + 2) - 3$ 42

20 Aufbau des Stacks für $a + 3 - 3$ 43

Abkürzungsverzeichnis

ANTLR	Another Tool for Language Recognition
ASCII	American Standard Code for Information interchange
AST	Abstract Syntax Tree
BASIC	Beginner's All-purpose Symbolic Instruction Code
BCEL	Byte Code Engineering Library
CYK	Cocke-Younger-Kasami
DEA	Deterministischer Endlicher Automat
EA	Endlicher Automat
EBNF	Erweiterte Backus-Naur-Form
IL	Intermediate Language
JVM	Java Virtual Machine
NEA	Nicht Deterministischer Automat
WYSIWYG	What You See Is What You Get
Yacc	Yet Another Compiler-Compiler

1 Einleitung und Aufgabenstellung

Nicht selten baut man nach Erlernen der ersten, zweiten oder vielleicht nach weiteren Programmiersprachen eine Vorstellung davon auf, wie die Sprache aussehen müsste, an der man weder etwas ergänzen noch weglassen könnte. Die Gradwanderung besteht darin, die Sprache so zu gestalten, dass sie einerseits einfach zu erlernen, andererseits ausdrucksstark genug ist. Die Vergangenheit hat gezeigt, dass es nicht immer erstrebenswert ist, gänzlich neue Paradigmen und Konzepte zu erfinden. Viel mehr waren die erfolgreichen Sprachen diejenigen, die Altes und Neues geschickt miteinander kombinierten. C++ entstand aus dem Bedarf heraus, die objektorientierten Konzepte aus Simula mit denen einer Sprache zum Schreiben effizienter Quelltexte auf unterster Ebene zu kombinieren. Das letztere Sprache C sein würde, ist nicht verwunderlich — schließlich saß Bjarne Stroustrup, Schöpfer von C++, im Bell Labs Computer Science Research Center neben den Unix-Pionieren und C-Entwicklern wie Dennis Ritchie und Brian Kerningham.

Ein Compiler übersetzt die Wörter einer Programmiersprache in die Wörter einer anderen Sprache, ohne die Bedeutung zu verändern. Die nötigen Grundlagen von der Definition bis hin zur Realisierung einer Programmiersprache liefert die Disziplin des Compilerbaus. Neben dem eigentlichen Entwurf eines Compilers kann diese Thematik insbesondere hilfreich beim Verstehen von Programmiersprachen sein. Auch finden sich die Techniken in vielen weiteren Teilgebieten der Software-Entwicklung wieder. Man denke bspw. an Suchfunktionen in Text-Editoren.

Was die Disziplin des Compilerbaus nur teilweise oder gar nicht beantworten kann, sind Entwurfsentscheidungen, die später den Umgang mit einer Sprache vereinfachen oder aber auch erschweren können: Soll ein Semikolon zum abschließen (C) oder trennen (Pascal) von Anweisungen dienen, oder vereinbart man eine Anweisung pro Zeile (BASIC)? Alle drei Varianten beeinflussen sicherlich nicht maßgebend die Effizienz eines Compilers, tragen aber zum Erscheinungsbild der Sprache bei.
Anhand der Programmiersprache BASIC soll dem Leser ein Einblick darüber verschafft werden, welche Überlegungen sich vor der formalen Definition einer Sprache, und damit auch vor Beginn der eigentlichen Techniken des Compilerbaus, aufkommen:
Thomas Kurtz und John Kemeny entwickelten 1963 die universelle Programmiersprache BASIC (Beginner's All-purpose Symbolic Instruction Code). Heute ist vor allem Visual Basic als einer von vielen Dialekten bekannt; jedoch ohne das Konzept der Zeilennummern fortzuführen oder den Einsatz von GOTO-Anweisungen zu fördern. In [BW09] nennt Kurtz u. a. folgende Überlegungen, die in den Entwurf von BASIC einflossen:

- Eine Zeile, eine Anweisung: *„Wir konnten keinen Punkt als Abschluss einer Anweisung nutzen [...] Und die Konvention mit einem Semikolon in Algol, so wie es auch der Fortran-Nachfolger C nutzte, erschien uns nicht sinnvoll.“*

- Zeilennummern sind GOTO-Ziele: *„Wir mussten Zeilennummern verwenden, da die Entwicklung in einer Zeit weit vor den Tagen von WYSIWYG*[1] *geschah.“*
- Eine Zahl ist eine Zahl ist eine Zahl: *„Es waren keine formellen Anforderungen bei der Eingabe einer Zahl im Code oder in Datenanweisungen vorhanden. Und die PRINT-Anweisung lieferte die Ausgabe in einem Standardformat. Die Anweisung FORMAT oder ihr Äquivalent in anderen Sprachen ist recht schwierig zu erlernen. Und der Anfänger wundert sich vielleicht, warum er sich überhaupt damit beschäftigen muss – er wollte doch eine ganz einfache Antwort haben!“*

Auf die Frage, ob Variablen in BASIC deklariert werden mussten, antwortete Kurtz in [BW09] weiterhin:

> *„Nein, überhaupt nicht. Tatsächlich waren Arrays immer einzelne Zeichen, gefolgt von der linken Klammer, sodass es eigentlich Deklarationen waren [. . .] Wenn Sie ein Array verwendeten, zum Beispiel durch `a(3)`, war das automatisch ein Array von, ich glaube 0 bis 10. Also eine automatische Standarddeklaration.“*

Solch eine Entwurfsentscheidung kann darüber entscheiden, ob BASIC nur für kleine Programme von Studenten oder aber auch Einsatz in größeren Programmen, wie für die Finanzbuchhaltung, zum Einsatz kommt. Ein falsch geschriebener Variablenbezeichner ist in einem kleinen Programm schnell ausfindig zu machen; in einem großen Programm kann das viel Zeit in Anspruch nehmen. Der BASIC-Compiler deklariert und initialisiert schließlich die falsche Variable automatisch und der Programmierer erhält damit keine Fehlermeldung.

1.1 Aufgabenstellung

Ziel dieser Studienarbeit ist es, in das Thema des Compilerbaus einzuführen. Darauf aufbauend sollen daraus Inhalte für die Lehrveranstaltung Informatik im Rahmen der Berufsakademie Gera abgeleitet werden können. Das schließt kurze Beispiele zum jeweiligen Thema ein.

[1] WYSIWYG ist die Abkürzung für „What You See Is What You Get“

2 Übersicht Kompilierung

Seit Aufkommen der ersten Compiler wie Fortran (unter Leitung von John W. Backus) in den frühen fünfziger Jahren, wurden die Grundlagen geschaffen, um systematisch die verschiedenen Aufgaben eines Compilers implementieren zu können. Dazu Zählen u. a. Pionierarbeiten von Backus zur formalen Notation von Sprachen oder von Kleene zum Thema der regulären Sprachen, die sich noch im weiteren Teil der Studienarbeit als besonders geeignet im Bereich der lexikalischen Analyse erweisen werden.
Nach [ASU99] besteht der Prozess des Kompilierens im Wesentlichen aus zwei Teilen:

1. *Analysephase*: Hier wird die Struktur wie ein Bezeichner, ein Literal oder eine Schleife des Quelltextes erkannt und daraus eine für die weitere Verarbeitung geeignete Zwischendarstellung aufgebaut.

2. *Synthesephase*: Hier werden aus der Zwischendarstellung Wörter in der jeweiligen Zielsprache, wie Maschinensprache oder Byte-Code, erzeugt.

Wiederum besteht die Analysephase aus drei Teilen:

1. *Lineare Analyse*: Einzelne Zeichen werden zu zusammenhängenden Symbolen zusammengefasst, wie Bezeichner oder Literale. Ein Symbol ist nur ein Name einer lexikalischen Einheit wie „Bezeichner“. Als Token wird die Kombination aus Symbol und Attributen (bspw. der Wert des Literals oder die Zeilennummer des Vorkommens) bezeichnet.

2. *Hierarchische Analyse*: Die Folge von Tokens, der sogenannte „Token-Stream“ wird hierarchisch zu verschachtelten Gruppen zusammengefasst. Eine Baumstruktur wird dabei aufgebaut. Beispielsweise bildet ein binärer Operator die Wurzel und dessen zwei Operanden die Kinder eines Teilbaums.

3. *Semantische Analyse*: Hier werden zusätzliche Regeln geprüft, die sich nicht allein aus der formalen Strukturbeschreibung der Quellsprache ergeben.

In Abbildung 1 sind die wesentlichen Bestandteile eines Compilers dargestellt, die im Folgenden vorgestellt werden.

2.1 Lexikalische Analyse

Die lineare Analyse wird beim Kompilieren als *„lexikalische Analyse“* (engl. Scanning) und der dazu verantwortliche Teil des Compilers als „Scanner“ oder „Lexer“ (kurz für Lexikalischer Scanner) bezeichnet. Der Scanner liest die Folge von Zeichen aus dem Quelltext ein und erzeugt daraus einen Token-Stream.

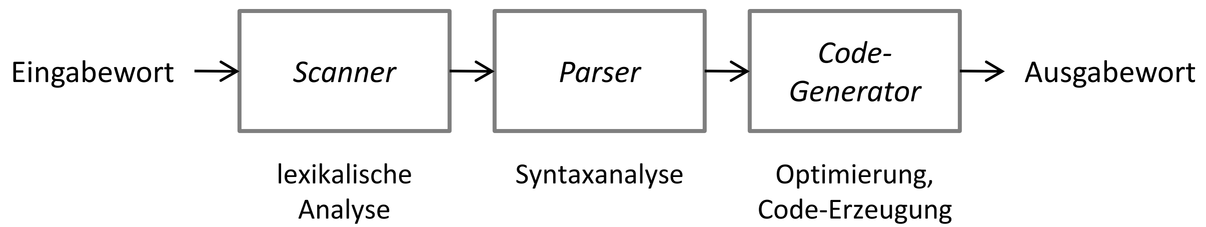


Abbildung 1: Wesentliche Bestandteile eines Compilers

Beispiel Die Zeichenfolge

```
printf("Hallo Welt\n");
```

würde nach der lexikalischen Analyse folgenden Token-Stream erzeugen:

1. der Bezeichner `printf`
2. die öffnende, runde Klammer `(`
3. das Stringliteral `Hallo Welt\n`
4. die schließende, runde Klammer `)`
5. das Semikolon `;`

Zur formalen Definition der Sprache von Symbolen eignen sich reguläre Ausdrücke. Aus den regulären Ausdrücken lassen sich endliche Automaten ableiten, die dann vom Scanner simuliert werden.

2.2 Syntaxanalyse

Als *„Syntaxanalyse“* (engl. Parsing) bezeichnet man beim Kompilieren die hierarchische Analyse und der dazugehörige Teil des Compilers als „Parser“. Tokens werden während der Syntaxanalyse zu grammatikalischen Sätzen zusammengefasst. Das geschieht nach den Produktionsregeln der Grammatik, deren Anwendung auf das Eingabewort (Quelltext) als „Ableitung“ bezeichnet wird. Die Ableitung lässt sich als Baum darstellen, den man als „Ableitungsbaum“, „Syntaxbaum“ oder engl. „Parse-Tree“ bezeichnet.

Beispiel Folgender Quelltextauszug

```
a = b + 10;
```

würde nach der Syntaxanalyse den in Abbildung 2 dargestellten Ableitungsbaum erzeugen. Die dazugehörigen Produktionsregeln lauten informell wie folgt:

1. Eine Anweisung ist eine Zuweisung, gefolgt von einem Semikolon.

2. Eine Zuweisung ist ein Bezeichner, gefolgt von einem Gleichheitszeichen und einem Ausdruck.

3. Ein Ausdruck ist ein Term, optional gefolgt von einem Plus- oder Minuszeichen und einem weiteren Term.

4. Ein Term ist ein Bezeichner oder ein Literal.

Für die weitere Verarbeitung haben die einzelnen Zeichen, wie das Gleichheitszeichen, im Ableitungsbaum keinen weiteren Nutzen mehr – die Zuweisung wurde schließlich bereits erkannt. Aus diesem Grund bauen viele Compiler nur einen logischen Baum, den sogenannten „abstrakten Syntaxbaum", engl. Abstract Syntax Tree – oft mit AST abgekürzt, auf.

Beispiel Folgende Struktur stellt Knoten für eine Zuweisung innerhalb eines ASTs dar. Das Gleichheitszeichen aus der konkreten Syntax wird dabei nicht gespeichert.

```
struct zuweisung {
    char bezeichner[255]; // linke Seite
    struct ausdruck;      // rechte Seite
};
```

Für die formale Definition der Produktionsregeln hat sich die EBNF-Notation (Erweiterte Backus-Naur-Form) durchgesetzt. Daraus lassen sich unterschiedliche Parser-Typen wie ein LL-Parser (Eingabe wird von links nach rechts gelesen und eine Linksableitung berechnet) erzeugen.

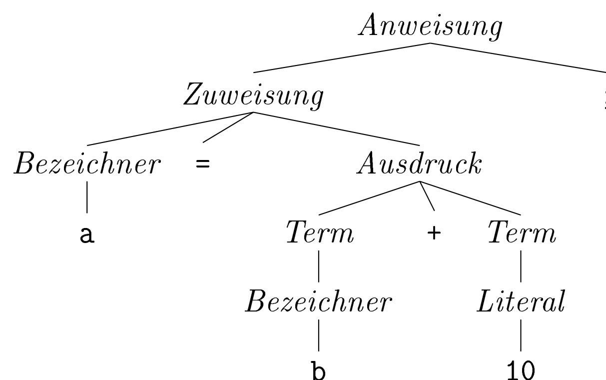


Abbildung 2: Ableitungsbaum für `a = b + 10;`

2.3 Semantische Analyse

Einerseits muss ein Compiler erkennen, ob ein Eingabewort (Quelltext) den strukturellen Bedingungen einer Sprache entspricht, andererseits sind weitere Bedingungen an das

Eingabewort geknüpft, die sich nur aus dem Kontext des Eingabewortes ergeben. Wirth berichtet in [Wir96], warum Programmiersprachen zwar kontextfreie Sprachen sind, aber die semantische Analyse den Kontext nicht komplett außer Acht lassen darf:

> *„Obwohl Programmiersprachen auf einer kontextfreien Syntax im Sinne von Chomsky[2] aufgebaut sind, so sind sie in einem allgemeineren Sinn doch kontextabhängig. Der hier angesprochene Kontextbezug besteht in der Tatsache, daß jeder Bezeichner vereinbart wird. In Vereinbarungen (Deklarationen) wird der Bezeichner einem Objekt zugeordnet, und zudem werden dem Objekt Attribute zugewiesen, wie z. B. ein Datentyp. Ein im Programmtext auftretender Bezeichner nimmt daher Bezug auf seine Vereinbarung, die außerhalb des syntaktischen Konstrukts liegt, in dem der Bezeichner auftritt. Wir sagen, seine Vereinbarung liege im Kontext des ihn enthaltenden Konstrukts."*

In vielen Sprachen muss bei einer Zuweisungen der Typ des Bezeichners der linken Seite mit dem Typ des Ausdrucks auf der rechten Seite übereinstimmen (abgesehen von der impliziten Abbildungen eines Typs, wie `short`, auf ein Typ mit einem größeren Wertebereich, wie `int`). Doch weder in der Umgebung der Deklaration dieses Bezeichners, noch in der Umgebung der Zuweisung kann diese Typprüfung syntaktisch stattfinden.

Beispiel Folgender Quelltextauszug ist syntaktisch einwandfrei aber nach den Regeln der statischen Semnatik von C falsch:

```
int addiere(int a, int b);
...
int x = addiere(10, 20, 30);
```

Nach den Regeln der Syntax von C können beliebig viele Argumente beim Aufruf einer Funktion übergeben werden; somit ist der Aufruf der Funktion `addiere` aus dem Beispiel mit *drei* Argumenten korrekt. Aber die Regeln der statischen Semantik besagen, dass der Funktion genauso viele Argumente wie vereinbarte Parameter zu übergeben sind; somit sind *drei* statt den vereinbarten *zwei* Argumenten falsch.

Knuth erweiterte 1968 hierzu die formalen Grammatiken um Attribute zu den attributierten Grammatiken. Der Ableitungsbaum wird dabei um Attribute dekoriert. So kann bspw. die Anzahl an Parametern einer Funktion in einem Attribut festgehalten und während der Übersetzung des Funktionsaufrufs herangezogen werden. Es sei noch angemerkt, dass die Attribute nicht ausschließlich Verwendung beim Prüfen der Semantik finden, sondern bspw. auch beim Allokieren von benötigtem Speicher einer Variablen in Abhängigkeit von dessen Typ.

[2] Die Chomsky-Hierarchie unterteilt formale Sprachen nach ihrer Mächtigkeit.

2.4 Synthesephase

In der Synthesephase findet, neben der Codeerzeugung, auch die Optimierung statt. Die Optimierung hat das Ziel, ohne die Semantik des Eingabewortes zu ändern, die Laufzeit und den Speicherbedarf des Ausgabewortes zu verbessern. Sie kann jeweils vor und nach der Code-Erzeugung durchgeführt werden; wobei erstere den AST manipuliert und letztere das Ausgabewort in der Zielsprache. Bei der Wahl der Optimierungsstrategie ist darauf zu achten, dass sich die Kompilierzeit nicht wesentlich durch die Optimierung verlangsamt. Desweiteren können auch Situationen auftreten, bei denen nach der Optimierung die Effizienz sich gar verschlechtert.

Beispiel Folgender Quelltextausschnitt soll zwei von vielen weiteren Optimierungsverfahren demonstrieren. Die Optimierung findet dabei vor der Codeerzeugung statt.

```
for (int i = 0; i < 1000; i++) {
   for (int j = 0; j < 5; j++) {
      // ...
   }
}
```

Es handelt sich hier um zwei verschachtelte Zählschleifen. Die Zählvariable `i` wird dabei einmal und die Zählvariable `j` 1000-mal initialisiert. Bei diesem Quelltext lässt sich die Verschachtelungsreihenfolge der beiden Schleifen austauschen, ohne dabei die Semantik zu ändern:

```
for (int j = 0; j < 5; j++) {
   for (int i = 0; i < 1000; i++) {
      // ...
   }
}
```

Nach Betrachten des umgestellten Quelltextes stellt sich heraus, dass die Zählvariable `j` einmal und die Zählvariable `i` nur noch fünfmal initialisiert werden muss, was zu einer Verbesserung der Laufzeit führt. Das allgemeine Verfahren lautet: Die Verschachtelungsreihenfolge von Schleifen ist aufsteigend anzuordnen so, dass die äußere Schleife die meisten und die innerste Schleife die wenigsten Iterationen durchläuft.
Ein anderes Verfahren sieht vor, dass eine Schleife mit sehr wenigen Iterationen und einem kurzen Schleifenkörper, als Sequenz umgeformt werden kann. Hintergrund dabei ist, dass bei jeder Iteration ein bedingter Sprung (Branch) zum Schleifenanfang erfolgen muss, was entgegen der Caching-Strategie der Zielmaschine arbeitet, denn bei jeden Sprung muss die Befehlsfolge erneut vom Hauptspeicher in den Cache geladen werden. Beim unmittelbaren Hintereinanderausführen des Schleifenkörpers kann so auf Sprünge verzichtet werden; allerdings steigt damit der Platzbedarf für die erzeugte Befehlsfolge.

Die innere Zählschleife aus dem Beispiel mit nur fünf Iterationen ist potenziell für dieses Optimierungsverfahren geeignet. Nach Anwenden des ersten Optimierungsverfahrens, bei dem der Schleifenkörper der einst inneren und nun äußeren Schleife angewachsen ist, könnte nun das Umformen der Schleife in eine Sequenz verhindern, da die Befehlsfolge des Schleifenkörpers zu lang ist.
Daraus lässt sich erkennen, dass die Wahl einer geeigneten Optimierungsstrategie, die mehrere Optimierungsverfahren miteinander kombiniert, eine nicht triviale Aufgabe ist.

Bis zu diesem Schritt arbeiten alle Teile eines Compilers unabhängig von der Zielsprache. Erst der nun folgende Schritt der Code-Erzeugung bindet an sie. Der entsprechende Teil des Compilers wird als „Code-Generator" bezeichnet. Er nimmt die Zwischendarstellung des Eingabewortes (z. B. den mit Attributen dekorierten AST) und bildet auf die Zielsprache ab.

Beispiel Folgender Quelltextauszug aus einem Java-Programm

```
for (int i = 0; i < 10; i++) { }
```

wird vom Java-Compiler in folgenden Byte-Code kompiliert:

```
L1                  // i = 0;
 ICONST_0           // Push 0
 ISTORE 1           // Pop in lokale Variable 1 (also i)
L2
 GOTO L3            // springe zu L3
L4                  // i++;
 FRAME APPEND [I]   // ohne Erlaeuterung
 IINC 1 1           // i = i + 1
L3                  // i < 10;
 FRAME SAME         // ohne Erlaeuterung
 ILOAD 1            // Push lokale Variable 1
 BIPUSH 10          // Push 10
 IF_ICMPLT L4       // Pop, Pop und wenn i < 10 Springe zu L4
```

Die Zielplattform für Java-Byte-Code ist die Java Virtual Machine (JVM), die nach dem Rechenmodell der Stack-Maschine arbeitet. Das gleiche Prinzip findet Anwendung bei der .NET-Software-Plattform mit IL-Code (Intermediate Language) als Zielsprache.
Die Übersetzung in Maschinencode für die weitverbreiteten x86-Prozessor-Architekturen muss hingegen auf Basis einer Register-Maschine stattfinden.

Nach der Code-Erzeugung findet eine zweite Optimierung statt, mit der Absicht das Ausgabewort zu optimieren. Meist ist das eine sogenannte „Peephole-Optimization" (engl. für Guckloch). Hier wird nur ein kleiner Ausschnitt (Guckloch) aus der erzeugten Befehlsfolge betrachtet. Dabei kommen Algorithmen zur Mustererkennung zum Einsatz.

Beispiel Folgender Byte-Code-Ausschnitt

```
BIPUSH 123 // a = 123;
ISTORE_1
ILOAD_1    // b = a;
ISTORE_2
```

kann durch folgenden, effizienteren Byte-Code ersetzt werden:

```
BIPUSH 123
DUP
ISTORE_1
ISTORE_2
```

Der Ausgangs-Code lädt den Wert `123` auf den Stack, speichert ihn in der Variablen `a` (Index `1`), lädt unmittelbar danach den Wert der Variablen `a` wieder auf den Stack und speichert den Wert in der Variablen `b` (Index `2`).
Der optimierte Code macht das Laden des Wertes aus dem Speicher der Variablen `a` überflüssig, in dem er nach dem Laden des Wertes `123` den Wert auf dem Stack mittels `DUP` dupliziert.

3 Grundlagen Wort und Sprache

Es werden hier Definitionen und Beispiele zu den Begriffen „Wort“ und „Sprache“ aufgeführt. Die Definitionen sind aus [Lan06] und [ASU99] entnommen. Dabei ist zu Beachten, dass wenn von einem „Wort“ die Rede ist, nicht der Alltagsbegriff gemeint ist, sondern ein Wort durchaus beliebig viele Leerräume enthalten kann, wenn es das Alphabet zulässt. So kann ein Quelltext als ein Wort aufgefasst werden.
Die Menge der natürlichen Zahlen $\mathbb{N}$ beginnt in dieser Studienarbeit bei der Eins. Soll die Null dazugehören, wird das Zeichen $\mathbb{N}_0$ verwendet.

3.1 Wort

Definition Ein *Alphabet* ist eine endliche, nichtleere Menge von Zeichen.

Beispiel Folgende sind geläufige Alphabete: Das binäre Alphabet $\Sigma = \{0, 1\}$, das Morse-Alphabet $\Sigma = \{., -, Pause\}$ oder das Computer-Alphabet ASCII (American Standard Code for Information Interchange).

Definition Sei A ein Alphabet. Ein *Wort* (eine Zeichenreihe) w über A ist eine endliche Folge

$w = w_0 \ldots w_{n-1}$ mit $w_i \in A, n \in \mathbb{N}$

$|w| = n$ ist die *Länge* des Wortes w.

Beispiel $A = \{a, \ldots, z\}$, $w = \mathit{for}$ damit ist $w_0 = f$, $w_1 = o$, $w_2 = r$ und $|w| = 3$.
Das Wort der Länge null wird mit ϵ bezeichnet, es wird das *leere Wort* genannt und damit ist $|\epsilon| = 0$.

Definition Die Menge aller Wörter über einem Alphabet A wird mit A^* (Kleenesche Hülle) bezeichnet:

$A^* = \{w \mid w = w_0 \ldots w_{n-1}, w_i \in A, n \in \mathbb{N}_0\}$.

Die Menge aller *nichtleeren Wörter* über A wird mit A^+ (positive Hülle) bezeichnet:

$A^+ = \{w \mid w = w_0 \ldots w_{n-1}, w_i \in A, n \in \mathbb{N}\} = A^* \setminus \{\epsilon\}$.

Definition Seien $x = x_0 \ldots x_{n-1}$ und $y = y_0 \ldots y_{m-1}$ Wörter über einem Alphabet A. Die *Verkettung* xy von x und y ist definiert als das Wort

$xy = x_0 \ldots x_{n-1} y_0 \ldots y_{m-1}$,

d. h. $(xy)_i = x_i$ falls $i \in \{0, \ldots, n-1\}$ bzw. y_{i-n} sonst.
Weiterhin gilt: $\epsilon x = x \epsilon = x$ für alle Wörter x.

Beispiel $x = for$, $y = each$, $xy = foreach$

Definition Die *i-te Potenz* eines Wortes x ist definiert durch:
$x^0 = \epsilon$,
$x^i = x^{i-1}x$ für alle $i \in \mathbb{N}$.

3.2 Sprache

Definition Sei A ein Alphabet. Eine Teilmenge $L \subseteq A^*$ heißt *formale Sprache* oder kurz *Sprache* über A.

Beispiel Es sei $A = \{a, b, c\}$. Dann sind bspw. folgende Mengen Sprachen über A:
$L_1 = \{aa, ab, aabcaacb\}$
$L_2 = \{\epsilon\}$
$L_3 = \emptyset$
$L_4 = \{a^n b^n c^n \mid n \in \mathbb{N}_0\}$
$L_5 = A^* \setminus \{abc\}$

Definition Die *i-te Potenz* einer Sprache X ist definiert durch:
$X^0 = \{\epsilon\}$,
$X^i = X^{i-1}X$ für alle $i \in \mathbb{N}$.

Definition Seien X und Y Sprachen. Die *Vereinigung* $X \cup Y$ von X und Y ist definiert durch:
$X \cup Y = \{w \mid w \text{ ist in } X \text{ oder } w \text{ ist in } Y\}$

Definition Seien X und Y Sprachen. Die *Konkatenation* XY von X und Y ist definiert durch:
$XY = \{xy \mid x \text{ ist in } X \text{ und } y \text{ ist in } Y\}$

Definition Sei X eine Sprache. Der *Kleene-Abschluss* X^* von X ist definiert durch:
$X^* = \bigcup_{i \in \mathbb{N}_0} X^i = \{\epsilon\} \cup X \cup X^2 \cup \ldots$

Definition Sei X eine Sprache. Der *positive Abschluss* X^+ von X ist definiert durch:
$X^+ = \bigcup_{i \in \mathbb{N}} X^i = X \cup X^2 \cup \ldots$

4 Lexikalische Analyse

Die Aufgabe der lexikalischen Analyse besteht darin, die Zeichenfolge des Eingabewortes in zusammenhängende Symbole zu gruppieren. Als Ausgabe entsteht eine Folge von Tokens, engl. *Token-Stream*, die als Eingabe der anschließenden Syntaxanalyse dient. Der entsprechende Teil des Compilers heißt „Scanner“ bzw. „Lexer“ (kurz für Lexikalischer Scanner). Neben der lexikalischen Analyse übernimmt ein Scanner typischerweise weitere Aufgaben wie:

- Entfernen von Kommentaren
- Entfernen von überflüssigen Leerräumen
- Aufbau einer Symboltabelle (bestehend aus Bezeichnern und deren Typen)
- Vorberechnen von Literalen, bspw. die Abbildung einer Ziffernfolge eines ganzzahligen Literals in eine computerinterne Ganzzahldarstellung

In [ASU99] wird die Trennung zwischen lexikalischer und syntaktischer Analyse als „etwas willkürlich“ bezeichnet. Die Trennung hat aber u. a. folgende Vorteile [ASU99]:

- Wenn der Scanner für die oben genannten Aufgaben, wie das Ignorieren von Kommentaren, zuständig ist, kann der Parser wesentlich einfacher aufgebaut werden.
- Scanner und Parser können voneinander getrennt optimiert werden. Beispielsweise können im Scanner Puffer-Techniken eingesetzt werden, um nicht aufwändig jedes Zeichen einzeln des Eingabewortes zu lesen.
- Der Compiler lässt sich besser portieren. Die Betriebssysteme Windows, Linux und MacOS unterscheiden sich in der Codierung eines Zeilenumbruchs. Der Scanner könnte an die jeweilige Codierung angepasst werden, ohne dass dabei der Parser geändert werden müsste.

Endliche Sprachen lassen sich durch Aufzählen ihrer Wörter angeben. Unendliche Sprachen hingegen erfordern eine Menge an Beschreibungen, die zusammen als „Syntax“ bezeichnet werden. Diese Syntax lässt sich informell oder formal angeben. Die wichtigsten Möglichkeiten, Syntaxen formal zu beschreiben, sind die Grammatiken, die erkennenden Automaten und die regulären Ausdrücke [Lan06]. Die Mächtigkeit einer Sprache ist ausschlaggebend darüber, welche Beschreibungsmöglichkeit geeignet ist, daher muss zunächst festgestellt werden, zu welcher Mächtigkeit die Sprache der Symbole zählt. In [ASU99] wird dazu angegeben:

„Bei lexikalischen Konstrukten ist Rekursion nicht nötig, im Gegensatz zu vielen syntaktischen Konstrukten."

In der sogenannten „Chomsky-Hierarchie", die Sprachen und dessen Grammatiken nach ihrer Beschränktheit unterteilt, sind die regulären Sprachen die beschränktesten. Mit ihnen lassen sich bspw. keine gepaarten, verschachtelten Konstrukte beschreiben. Die Einschränkung ist aber hinreichend zum Beschreiben von lexikalischen Konstrukten. Im Folgenden werden daher die regulären Sprachen und die regulären Ausdrücke und regulären Sprachen betrachtet.

4.1 Reguläre Ausdrücke und reguläre Sprachen

Ein Pascal-Bezeichner kann informell beschrieben werden durch:

„Ein Buchstabe gefolgt von beliebig vielen Buchstaben oder Ziffern"

Die Sprache ist unendlich, da alle Bezeichner beliebig lang (mit Ausnahme von null) sein dürfen. Da alle Bezeichner nach einem Muster aufgebaut sind, kann die Beschreibung (das Muster) in endlicher Länge erfolgen. Diese Beschreibung kann formal durch die Notation der *regulären Ausdrücke* beschrieben werden. Für das Beispiel der Pascal-Bezeichner lautet der reguläre Ausdruck:

```
letter(letter|digit)*
```

Der `|` wird als „oder" gelesen, die Klammern dienen zum Gruppieren von Teilausdrücken und der `*` ist der Kleene-Abschluss und wird als „beliebig oft" gelesen.

Die von einem regulären Ausdruck r bezeichnete Sprache L, formal als $L(r)$ geschrieben, wird als „reguläre Menge" bezeichnet [ASU99].

Definition Sei Σ ein Alphabet. Die *regulären Ausdrücke* über Σ und die von ihnen beschriebenen Mengen werden wie folgt rekursiv definiert [HU00]:

1. $\emptyset$ ist ein regulärer Ausdruck und bezeichnet die leere Menge.
2. ϵ ist ein regulärer Ausdruck und bezeichnet die Menge $\{\epsilon\}$.
3. Für jedes Zeichen a aus Σ ist a ein regulärer Ausdruck und bezeichnet die Menge $\{a\}$ (ob es sich um ein Zeichen oder um einen regulären Ausdruck handelt, muss aus dem Kontext hervorgehen).
4. Wenn r und s reguläre Ausdrücke sind, die die Sprachen R und S bezeichnen, so sind $(r|s)$, (rs) und (r^*) ebenfalls reguläre Ausdrücke und bezeichnen die Mengen $R \cup S$, RS bzw. R^*.

Zudem gibt es häufig verwendete Abkürzungen, um reguläre Ausdrücke kompakter zu notieren. Einige davon werden in [ASU99] aufgeführt:

1. *Ein- oder mehrfaches Auftreten*: Der einstellige Postfix-Operator $^+$ bedeutet, dass der davorstehende Ausdruck einmal oder öfters vorkommt. Sei r ein regulärer Ausdruck, dann ist der reguläre Ausdruck r^+ gleichbedeutend mit rr^*. Es gilt also: $r^+ = rr^*$ bzw. $r^* = r^+|\epsilon$. Der Operator $^+$ hat dieselbe Priorität und Assoziativität wie der Operator *.

2. *Null- oder einmaliges Auftreten*: Der einstellige Postfix-Operator ? bedeutet, dass der davorstehende Ausdruck keinmal oder einmal vorkommt. Sei r ein regulärer Ausdruck, dann ist der reguläre Ausdruck $r?$ eine Abkürzung für $r|\epsilon$. Es gilt also: $r? = r|\epsilon$.

3. *Zeichenklassen*: Der reguläre Ausdruck $[abc]$, wobei a, b und c Zeichen des Alphabets sind, bezeichnet den regulären Ausdruck $a|b|c$. Desweiteren gibt es abgekürzte Zeichenklassen wie $[a-z]$, die den regulären Ausdruck $a|b|\ldots|z$ beschreibt.

Beim Schreiben regulärer Ausdrücke können Klammern gespart werden, wenn folgende Konventionen getroffen werden [HU00]:

1. Die einstelligen Operatoren * und $^+$ haben die höchste Priorität und sind links-assoziativ.

2. Die Konkatenation hat die zweithöchste Priorität und ist links-assoziativ.

3. Die Vereinigung | hat die niedrigste Priorität und ist links-assoziativ.

Nach dieser Konvention ist bspw. $((a(b^*))|a)$ äquivalent zum vereinfachten Ausdruck $ab^*|a$.

Beispiel Sei A ein Alphabet mit $A = \{a, b\}$ und Folgende reguläre Ausdrücke über A.

1. Der reguläre Ausdruck a beschreibt die Menge $\{a\}$

2. Der reguläre Ausdruck $aa|b$ beschreibt die Menge $\{aa, b\}$ und *nicht* die Menge $\{aa, ab\}$, da die Konkatenation eine höhere Priorität besitzt als $|$.

3. Der reguläre Ausdruck a^* beschreibt die Menge $\{\epsilon, a, aa, \ldots\}$.

4. Der reguläre Ausdruck a^+b^+ ist eine Abkürzung für aa^*bb^* und beschreibt die Menge $\{ab, aab, aaab, abb, abb, aabb, \ldots\}$, also alle Wörter, die mit mindestens einem a beginnen und mit mindestens einem b enden.

5. Der reguläre Ausdruck $a|a^*b$ bezeichnet die Menge $\{a, b, ab, aab, aaab, \ldots\}$, also alle Wörter a, b oder die, die mit mindestens einem a beginnen und mit einem b enden.

Nach [ASU99] sind zwei reguläre Ausdrücke r und s *äquivalent*, wenn durch sie die gleiche Sprache beschrieben wird. Man schreibt dafür $r = s$. Beispielsweise ist $(a|b) = (b|a)$, da sie beide die Menge $\{a, b\}$ beschreiben. Um die Äquivalenz von regulären Ausdrücken beweisen zu können, gibt es eine Menge von algebraischen Gesetzen zum Umstellen von regulären Ausdrücken [ASU99]:

Axiom	Erläuterung
$r\|s = s\|r$	$\|$ ist kommutativ
$r\|(s\|t) = (r\|s)\|t$	$\|$ ist assoziativ
$(rs)t = r(st)$	die Konkatenation ist assoziativ
$r(s\|t) = rs\|rt$	
$(s\|t)r = st\|tr$	die Konkatenation ist distributiv bzwgl. $\|$
$\epsilon r = r$	
$r\epsilon = r$	ϵ ist das neutrale Element der Konkatenation
$r^* = (r\|\epsilon)^*$	Beziehung zwischen * und ϵ
$r^{**} = r^*$	* ist idempotent

Zur weiteren Vereinfachung der Schreibweise werden im Folgenden die *regulären Definitionen* eingeführt. Sie weisen regulären Ausdrücken Namen zu und können selbst wie Zeichen aus einem Alphabet verwendet werden [ASU99]:

$$\begin{aligned} &d_1 \to r_1 \\ &d_2 \to r_2 \\ &\quad\vdots \\ &d_n \to r_n \end{aligned}$$

Dabei ist d_i ein eindeutiger Name und r_i der dazugehörige reguläre Ausdruck über den Zeichen aus $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$. Wichtig ist, dass sich eine Definition d_i nur auf die Definitionen von d_1 bis d_{i-1} beziehen kann. Damit ist sichergestellt, dass sämtliche Referenzen auf vorherige Definitionen iterativ durch deren reguläre Ausdrücke ersetzt werden können (andernfalls könnten rekursive Konstrukte beschrieben werden).

Beispiel Für die nachfolgende, sehr einfache Programmiersprache

```
var i
var n

input j
for i = 1 to n
   print i
next
```

lautet die reguläre Definition für die lexikalischen Konstrukte:

$$\begin{aligned} var &\rightarrow var \\ &\vdots \\ next &\rightarrow next \\ digit &\rightarrow [0-9] \\ letter &\rightarrow [A-Z]|[a-z] \\ number &\rightarrow (+|-)?digit^{+} \\ identifier &\rightarrow letter(letter|digit)^{*} \end{aligned}$$

Nun würde die Definition von dem Schlüsselwort *var* auch auf die Definition von *identifier* zutreffen, denn es gilt $L(var) \subseteq L(letter(letter|digit)^{*})$. Daher wird vereinbart, dass die Definition Vorrang hat, die das längste Teilwort erkennt. So hätte *identifier* bei dem Teilwort *nextChar* Vorrang gegenüber *next*. Erkennen mehrere Definitionen die gleiche Länge, so hat die Definition Vorrang, die den kleineren Index besitzt.

4.2 Endliche Automaten

Ein *endlicher Automat* (EA) ist ein Modell zum Beschreiben von Verhalten. Er besteht aus einer *endlichen* Menge von Zuständen und Übergängen (oft auch als *Transitionen* bezeichnet). Aufgrund eines einzelnen gelesenen Zeichens kann er von einem Zustand in einen neuen Zustand wechseln. Dabei gibt es einen ausgezeichneten Zustand s_0 der als „Startzustand“ bezeichnet wird und in dem sich der EA anfangs befindet. Ist der EA ein *Akzeptor* (Erkenner), so sind einige Zustände als Endzustände ausgezeichnet. Befindet sich der EA nach Lesen des *kompletten* Eingabewortes in einem der Endzustände, wird das Wort akzeptiert ansonsten nicht.
Der Satz von Kleene besagt:

> *„Jede Sprache, die von einem endlichen Automaten akzeptiert wird ist regulär und jede reguläre Sprache wird von einem endlichen Automaten akzeptiert.“*

Damit eignen sich EAs zum Erkennen von lexikalischen Konstrukten. Es gibt Regeln, nach denen ein EA schrittweise aus einem regulären Ausdruck aufgebaut werden kann. Ein Scanner lässt sich durch Simulation eines solchen konstruierten EAs aufbauen.

Die erste Klasse von EAs, die im Weiteren betrachtet werden, sind die *deterministischen endlichen Automaten* (DEAs). Sie unterscheiden sich von den *endlichen nicht deterministischen Automaten* (NEAs) dahingehend, dass sie nur einen Zustandsübergang beim Lesen eines Zeichens ausführen können.

Definition Ein *deterministischer endlicher Automat* A ist ein Quintupel $A = (S, \Sigma, \delta, s_0, F)$ mit

- einer Zustandsmenge S
- einem Eingabealphabet Σ
- einer Übergangsfunktion δ, die $S \times \Sigma$ auf S abbildet, d. h. $\delta(s, a)$ ist ein Zustand für jeden Zustand s und jedes Eingabezeichen a mit $a \in \Sigma$
- einem Startzustand s_0 mit $s_0 \in S$, in dem sich der Automat anfangs befindet
- einer Menge akzeptierender Zustände (Endzustände) F mit $F \subseteq S$

Ein DEA lässt sich durch einen markierten, gerichteten Graphen, der als „Übergangsgraph" bezeichnet wird, darstellen. Dabei stellen die Knoten die Zustände und die markierten Kanten die Übergangsfunktion dar. Die Kanten sind mit den Zeichen aus dem Eingabealphabet markiert. Gibt es einen Zustandsübergang von s_u nach Lesen eines Zeichens a in $s_v = \delta(s_u, a)$, so gibt es eine mit a markierte Kante im Übergangsgraph von s_u nach s_v
Ein DEA akzeptiert ein Eingabewort dann, wenn es einen Pfad vom Startzustand in eines der Endzustände gibt, wobei die Kantenmarkierungen entlang dieses Pfades das Eingabewort bildet.

Beispiel Der DEA $(S, \Sigma, \delta, s_0, F)$ akzeptiert die Vergleichsoperatoren `==`, `<=` und `>=` mit

- der Zustandsmenge $S = \{0, 1, 2, 3, 4, 5\}$,
- dem Eingabealphabet $\Sigma = \{=, >, <\}$,
- der Übergangsfunktion δ aus Abbilding 3,
- dem Startzustand $s_0 = 0$ und
- den Endzuständen $F = \{3, 4, 5\}$.

Zustand	Eingabezeichen		
	=	>	<
0	1	-	2
1	3	4	-
2	5	-	-

Abbildung 3: Übergangsfunktion für den DEA aus Abbildung 4

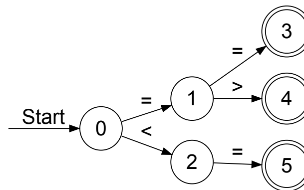


Abbildung 4: Ein DEA, der die Vergleichsoperatoren ==, <= und >= erkennt

Der Übergangsgraph ist in Abbildung 4 dargestellt.

Für die Konstruktion eines EAs aus einem regulären Ausdruck, der die gleiche reguläre Sprache erkennt, die der reguläre Ausdruck beschreibt, wird nun der DEA zu einem NEA mit ϵ-Übergängen erweitert (NEAs bilden eine Untermenge von NEAs). Der NEA als auch der DEA sind gleichmächtig beim Erkennen von regulären Sprachen, die Konstruktion eines NEAs mit ϵ-Übergängen aus einem regulären Ausdruck ist jedoch bequemer. Es ist aber immer möglich, einen DEA aus einem NEA mit ϵ-Übergängen zu konstruieren, der die gleiche Sprache akzeptiert.

Während der DEA von einem Zustand beim Lesen eines Eingabezeichens in nur *einen* (möglicherweise selben) Zustand wechseln kann, ist der NEA in der Lage, in *mehrere* Zustände zu wechseln. Stellt man sich gedanklich den NEA als eine Maschine vor, so entspräche der Übergang in mehrere Zustände dem Selbstklonen der Maschine für jeweils einen Übergangszustand. Jede geklonte Maschine setzt ihre Arbeit ab dem jeweiligen Zustand mit dem nächsten Eingabezeichen fort.
Die Übergangsfunktion δ wird für einen NEA von $\delta : S \times \Sigma \rightarrow S$ erweitert zu

$$\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$$

wobei $\mathcal{P}(S)$ die Potenzmenge von S darstellt.
Für einen NEA mit ϵ-Übergängen muss die Übergangsfunktion nochmals erweitert werden:

$$\delta : S \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(S)$$

Was bedeutet, dass es zusätzlich Übergänge für die leere Eingabe ϵ geben kann. Ein Eingabewort wird von einem NEA mit ϵ-Übergängen akzeptiert, wenn es einen Pfad vom Startzustand zu einem Endzustand gibt, und die Markierungen entlang dieses Pfades, ohne den Markierungen für ϵ, das Eingabewort bilden.

4.3 Konstruktion endlicher Automaten aus regulären Ausdrücken

Es wird nun ein Algorithmus vorgestellt, der als „Thompson-Konstruktion“ bezeichnet wird. Mit ihm lässt sich aus einem gegebenen regulären Ausdruck r ein NEA mit ϵ-Übergängen M konstruieren so, dass $L(r) = L(M)$ gilt. Die Überlegungen sind aus [HU00] entnommen:

1. *Konstruktion aus einfachen regulären Ausdrücken*: Für die regulären Ausdrücke $\emptyset$, ϵ und $a \in \Sigma$ ist es offensichtlich, dass die NEAs aus Abbildung 5 die gleichen Sprachen beschreiben.

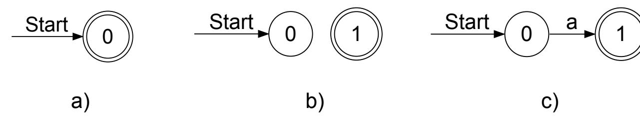


Abbildung 5: NEAs aus den regulären Ausdrücken a) ϵ, b) $\emptyset$ und c) $a \in \Sigma$

2. *Konstruktion aus der Vereinigung*: Es wird die Vereinigung $r|s$ von zwei regulären Ausdrücken betrachtet. Seien jeweils die beiden NEAs
   $M_1 = (S_1, \Sigma_1, \delta_1, s_1, \{f_1\})$ mit $L(M_1) = L(r)$ und
   $M_2 = (S_2, \Sigma_2, \delta_2, s_2, \{f_2\})$ mit $L(M_2) = L(s)$
   aus r und s konstruiert. Da sich die Zustände umbenennen lassen, kann angenommen werden, dass $S_1 \cap S_2 = \emptyset$ gilt, also die Zustandsmengen beider NEAs disjunkt sind. Für $r|s$ wird nun ein neuer NEA konstruiert:
   $M = (S_1 \cup S_2 \cup \{s_0, f_0\}, \Sigma_1 \cup \Sigma_2, \delta, s_0, \{f_0\})$
   wobei δ definiert ist durch

   i) $\delta(s_0, \epsilon) = \{s_1, s_2\}$

   ii) $\delta(s, a) = \delta_1(q, a)$ für $s \in S_1 \setminus \{f_1\}$ und $a \in \Sigma_1 \cup \{\epsilon\}$

   iii) $\delta(s, a) = \delta_2(q, a)$ für $s \in S_2 \setminus \{f_2\}$ und $a \in \Sigma_2 \cup \{\epsilon\}$

   iv) $\delta(f_1, \epsilon) = \delta(f_2, \epsilon) = \{f_0\}$

   In Abbildung 6 ist die Konstruktion von M dargestellt. Gedanklich wird beim Simulieren von M M_1 und M_2 parallel mit demselben Wort abgearbeitet. Erreicht M_1 den Endzustand f_1, so wird es dank $\delta(f_1, \epsilon) = \{f_0\}$ auch einen Zustandsübergang in den Endzustand f_0 von M geben. Das gilt analog für M_2 und f_2.

3. *Konstruktion aus der Konkatenation*: Seien M_1 und M_2 wie zuvor. Die Konstruktion von M für den regulären Ausdruck rs lautet:
   $M = (S_1 \cup S_2, \Sigma_1 \cup \Sigma_2, \delta, \{s_1\}, \{f_2\})$
   wobei δ definiert ist durch

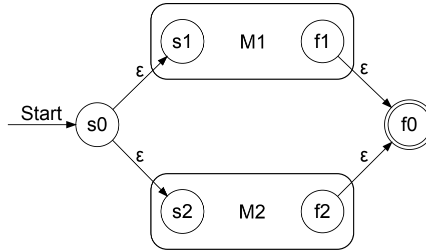


Abbildung 6: Konstruktion von M für $r|s$ aus M_1 und M_2

i) $\delta(s, a) = \delta_1(s, a)$ für $s \in S_1 \setminus \{f_1\}$ und $a \in \Sigma_1 \cup \{\epsilon\}$

ii) $\delta(s_1, \epsilon) = \{s_2\}$

iii) $\delta(s, a) = \delta_2(s, a)$ für $s \in S_2$ und $a \in \Sigma_2 \cup \{\epsilon\}$

In Abbildung 7 ist die Konstruktion von M dargestellt. Gedanklich wird beim Simulieren von M zuerst M_1 abgearbeitet. Erreicht M_1 seinen Endzustand, so wird dank $\delta(s_1, \epsilon) = \{s_2\}$ anschließend mit dem Abarbeiten von M_2 und dem verbleibenden Teilwort fortgeführt.

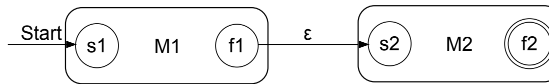


Abbildung 7: Konstruktion von M für rs aus M_1 und M_2

4. *Konstruktion aus der Hüllenbildung*: Es wird die Hüllenbildng r^* für den regulären Ausdruck r betrachtet. Sei der NEA

   $M_1 = (S_1, \Sigma_1, \delta_1, s_1, \{f_1\})$ mit $L(M_1) = L(r)$

   aus r konstruiert. Die Konstruktion von M aus r^* lautet:

   $M = (S_1 \cup \{s_0, f_0\}, \Sigma_1, \delta, \{s_0\}, \{f_0\})$

   wobei δ definiert ist durch

   i) $\delta(s_0, \epsilon) = \delta(f_1, \epsilon) = \{s_1, f_0\}$

   ii) $\delta(s, a) = \delta_1(s, a)$ für $s \in S_1 \setminus \{f_1\}$ und $a \in \Sigma_1 \cup \{\epsilon\}$

   In Abbildung 8 ist die Konstruktion von M dargestellt. Für den Fall, dass sich r keinmal auf das Eingabewort anwenden lässt, kann dank $\delta(s_0, \epsilon) = \{s_1, f_0\}$ direkt in den Endzustand übergegangen werden; andernfalls wird gedanklich M_1 abgearbeitet und dank $\delta(f_1, \epsilon) = \{s_1\}$ danach geprüft, ob sich r erneut anwenden lässt. Wenn nicht, dann wird dank $\delta(f_1, \epsilon) = \{f_0\}$ in den Endzustand übergegangen.

Algorithmus *Thompson-Konstruktion* eines NEAs mit ϵ-Übergängen aus einem regulären Ausdruck r.

Eingabe: Ein regulärer Ausdruck r über einem Alphabet Σ.

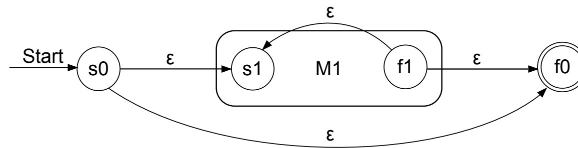


Abbildung 8: Konstruktion von M für r^* aus M_1

Ausgabe: Ein NEA mit ϵ-Übergängen M, der $L(r)$ akzeptiert.

Methode:

1. Es werden zunächst alle Abkürzungen wie r^+ oder $r?$ aufgelöst.
2. Der aufgelöste reguläre Ausdruck r wird in seine Bestandteile unter Berücksichtigung der Prioritäten von Operatoren und Klammern zerlegt.
3. Auf die Bestandteile werden die obigen Konstruktionen aus den einfachen regulären Ausdrücken, aus der Vereinigung, aus der Konkatenation und aus der Hüllenbildung angewandt. Dabei erhält jeder neu erzeugte Zustand einen neuen Namen, damit ausgeschlossen ist, dass zwei Zustände eines Teil-NEAs den gleichen Namen haben.

Beispiel Für den regulären Ausdruck $a(b|c)?$ soll der Algorithmus angewandt werden:

1. *Auflösen der Abkürzungen*: $r = a((b|c)|\epsilon)$
2. *Dekomposition des Ausdrucks*: Die Dekomposition von r ist in Abbildung 9 dargestellt.

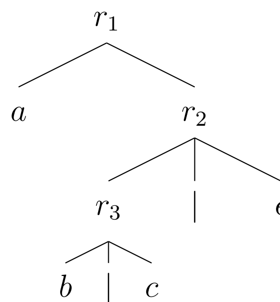


Abbildung 9: Dekomposition von $r = a((b|c)|\epsilon)$

3. *Anwenden der Konstruktionen*: In Abbildung 10 sind die konstruierten NEAs aus a, b, c, ϵ und $r_3 = b|c$ dargestellt. Für r_3 wurde die Regel zur Konstruktion aus der Vereinigung angewandt.

   Der konstruierte NEA aus r_2 setzt sich aus den NEAs für r_3 und ϵ zusammen. Er ist in Abbildung 11 dargestellt. Auch hier wurde die Regel zur Konstruktion aus der Vereinigung angewandt.

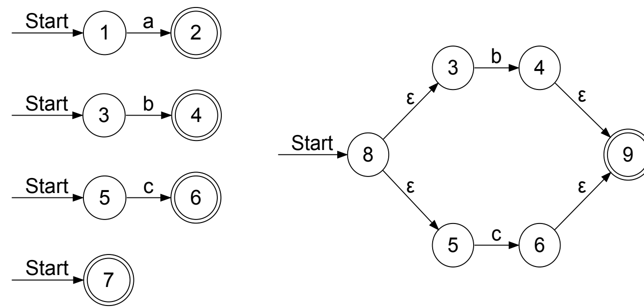


Abbildung 10: NEAs für a, b, c, ϵ (links) und $r_3 = b|c$ (rechts)

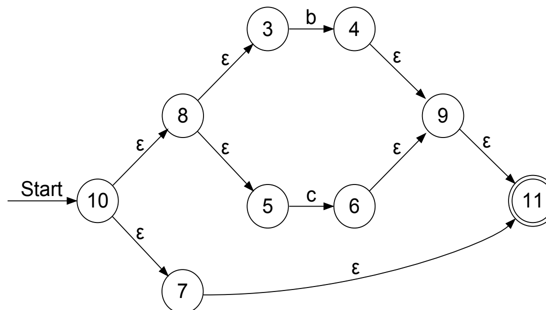


Abbildung 11: NEAs für $r_2 = (b|c)|\epsilon)$

Zum Schluss wurde der NEA für r_1 aus den NEAs für a und r_2 zusammengesetzt mit Anwendung der Regel für die Konkatenation. Der NEA für r_1 ist in Abbildung 12 dargestellt.

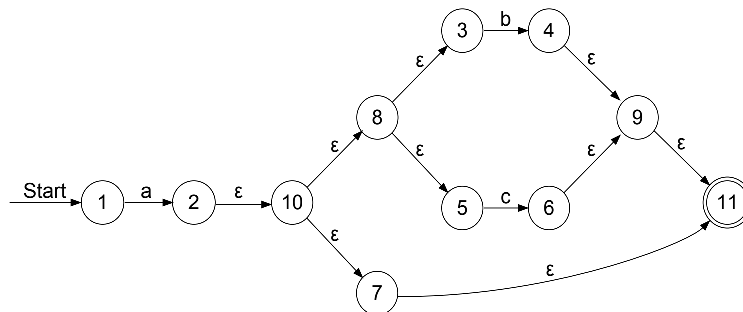


Abbildung 12: NEAs für $r_1 = a((b|c)|\epsilon)$

4.4 Simulation endlicher Automaten

Die grundlegende Aufgabe eines Scanners besteht darin, Symbole anhand ihrer Beschreibung, den regulären Ausdrücken, zu erkennen. Im vorherigen Abschnitt wurde mit der *Thompson-Konstruktion* eine Möglichkeit vorgestellt, aus regulären Ausdrücken NEAs mit ϵ-Übergängen systematisch zu konstruieren. Nun wird ein Verfahren aus [ASU99] gezeigt, mit dem man einen solchen NEA mit ϵ-Übergängen simulieren kann. Damit kann für ein konkretes Eingabewort entschieden werden, ob es Teil der vom NEA beschriebenen

regulären Menge ist.

Algorithmus Simulation eines NEAs.

Eingabe: Ein mit der vorhergehenden *Thompson-Konstruktion* konstruierter Automat M und ein Eingabewort w, dass mit dem Dateiendezeichen `eof` (End of File) abgeschlossen ist. Der Startzustand von M ist s_0 und die Menge der Endzustände ist F.

Ausgabe: Die Antwort „Ja“, wenn w von M akzeptiert wird, sonst „Nein“.

Methode:

$S := \epsilon{-}closure(\{s_0\})$
$a := nextchar()$
$while\ a \neq eof\ do\ begin$
  $S := \epsilon{-}closure(move(S, a))$
  $a := nextchar()$
end
$if\ S \cap F \neq \emptyset\ then$
  $return\ Ja$
$else\ return\ Nein$

mit

- $\epsilon{-}closure(s)$: Gibt die Menge aller Zustände t zurück, für die es vom Zustand s aus einen mit null oder mehr ϵ markierten Pfad von s nach t gibt.
- $\epsilon - closure(T)$: Gibt die Vereinigung von $\epsilon - closure(s)$ für alle Zustände s aus T zurück, also $\bigcup_{s \in T} \epsilon{-}closure(s)$.
- $move(T, a)$: Gibt die Menge aller Zustände zurück, die von einem Zustand s aus T nach Lesen des Eingabezeichens a erreicht werden können.

Zunächst wird die Zustandsmenge berechnet, die vom Startzustand über ein mit null oder mehr ϵ markierten Pfad erreichbar ist und in S gespeichert. Danach wird das erste Zeichen aus dem Eingabewort gelesen. Wurde noch nicht das Ende des Eingabewortes erreicht, dann werden alle Zustände T berechnet, die von S aus durch a erreichbar sind. Die Zustandsmenge S wird erneut berechnet mit $\epsilon - closure(T)$ und mit dem nächsten Zeichen fortgefahren.
Für jede Iteration enthält S also die Zustände, die von dem bis dahin gelesenen Präfix des Eingabewortes aus erreichbar sind. Nach dem vollständigen Lesen des Eingabewortes wird geprüft, ob in S mindestens ein Endzustand enthalten ist. Trifft das zu, dann wird „Ja“, sonst „Nein“ als Antwort gegeben.

Beispiel Für den NEA aus Abbildung 12 soll der Algorithmus auf das Eingabewort $w = ac\ eof$ angewandt werden:

1. $S = \epsilon{-}closure(\{0\}) = \{1\}$
   $a = a$
2. $S = \epsilon{-}closure(move(S, a)) = \epsilon{-}closure(\{2\}) = \{3, 5, 7, 8, 10, 11\}$
   $a = c$
3. $S = \epsilon{-}closure(move(S, a)) = \epsilon{-}closure(\{6\}) = \{9, 11\}$
   $a = eof$

Es gilt $S \cap F = \{9, 11\} \cap \{11\} = \{11\} \neq \emptyset$, somit wird das Eingabewort ac von diesem NEA akzeptiert. Der Pfad von 1 nach 11 für ac ist:

$$1 \xrightarrow{a} 2 \xrightarrow{\epsilon} 10 \xrightarrow{\epsilon} 8 \xrightarrow{\epsilon} 5 \xrightarrow{c} 6 \xrightarrow{\epsilon} 9 \xrightarrow{\epsilon} 11$$

Eine effiziente Implementierung dieses Algorithmus wird in [Lan06] angegeben, der leicht abgewandelt hier vorgestellt werden soll. Die Implementierung arbeitet mit drei Datenstrukturen: Zwei Schlangen P und Q sowie einem mit Zuständen indizierten Bitvektor M. Dabei enthält Q die Menge der aktuellen Zustände, P die Folgezustände aus Q für das aktuell gelesene Zeichen und in M sind alle in Q enthaltenen Zustände markiert, damit schnell geschaut werden kann, ob ein Zustand bereits in Q existiert bevor er in Q hinzugefügt wird. Die Prozedur zum Simulieren lautet:

```
markiere s0
enqueue s0 in Q
a = nextchar()
while a ≠ eof do begin
    ϵ−closure()
    a = nextchar()
    move()
end
if Q ∩ F ≠ ∅ then
    return Ja
else return Nein
```

mit $\epsilon{-}closure$:

```
while |Q| > 0 do begin
    s = dequeue aus Q
    enqueue s in P
    for each t ∈ Folgezustaende von s mit ϵ−Uebergang do begin
```

$if\ t\ unmarkiert\ then\ begin$
$markiere\ t$
$enqueue\ t\ in\ Q$
end
end
end

und *move*:

$while\ |P| > 0\ do\ begin$
$s\ =\ dequeue\ aus\ P$
$unmarkiere\ s$
$if\ \exists t\ mit\ t \in Folgezustaende\ von\ s\ mit\ a{-}Uebergang\ then\ begin$
$markiere\ t$
$enqueue\ t\ in\ Q$
end
end

Nach [ASU99] besitzt diese Variante einen Speicherbedarf von $O(|r|)$ und einen Zeitbedarf von $O(|r| \times |w|)$.
Die Alternative wäre einen DEA aus dem NEA zu konstruieren und diesen zu simulieren. Diese Variante würde nach [ASU99] einen Speicherbedarf von $O(2^{|r|})$ und einen Zeitbedarf von $O(|w|)$ besitzen.

4.5 Entwurf eines Scanners

Mit diesem Vorwissen lässt sich nun ein Scanner für lexikalische Konstrukte entwerfen. Die Beschreibung der Konstrukte (Bezeichner, Literale, Operatoren etc.) erfolgt mit Hilfe der regulären Definition:

$$d_1 \to r_1$$
$$d_2 \to r_2$$
$$\vdots$$
$$d_n \to r_n$$

Aus ihr lässt sich für jede Definition d_i, also einem regulären Ausdruck, systematisch durch die Thompson-Konstruktion ein Teil-NEA konstruieren. Der Scanner muss nun diese Teil-NEAs für das Eingabewort (Quelltext) simulieren. Es wird dazu ein NEA wie in Abbildung 13 aus allen Teil-NEAs konstruiert, so dass der Startzustand jeweils einen ϵ-Übergang zu den einzelnen Teil-NEAs besitzt.

Desweiteren muss der oben vorgestellte Algorithmus zum Simulieren eines NEAs noch leicht abgeändert werden, damit das längste Präfix, dass auf eine Definition passt, er-

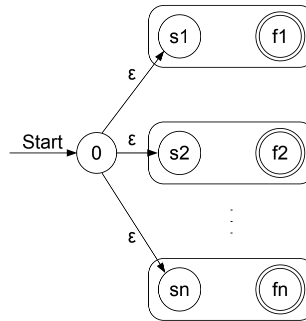


Abbildung 13: Entwurf eines Scanners aus mehreren Teil-NEAs

kannt wird [ASU99]:
Bei der Thompson-Konstruktion gibt es jeweils nur einen Endzustand für eine Definition und dieser besitzt keine Zustandsübergänge. Damit kann der NEA das Eingabewort so lange abarbeiten, bis kein Zustandsübergang mehr möglich ist (Terminierung). Damit ist sichergestellt, dass sich der NEA nach Terminierung in einen der Endzustände befindet, deren Definition das längste Präfix erkannt hat. Wurde das gleiche Präfix nach Terminierung von mehreren Definitionen erkannt, dann wird das von ihnen zuerst definierte ausgewählt. Danach erfolgt der jeweilige Eintrag in den Token-Stream und es werden ggf. weitere verknüpfte Aktionen (bspw. Abbildung einer Ziffernfolge für Ganzzahlliterale in eine computerinterne Ganzzahldarstellung) ausgelöst.

Abschließend sei noch auf zwei Dinge hingewiesen: Es existieren Algorithmen, die ein gegebenen EA bzgl. der Anzahl an Zuständen minimieren können. Beispielsweise kann man einen solchen Algorithmus zur Minimierung eines DFAs aus dem *Satz von Myhill-Nerode* ableiten. Weiterhin sei auf Scanner-Generatoren hingewiesen, wie das in der Unix-Welt weitverbreitete *Lex* von Lesk, die aus einer Beschreibungssprache für lexikalische Konstrukte *automatisch* einen Scanner generieren können. Sie basieren mehr oder weniger auf den oben genannten Verfahren zur Konstruktion und Simulation.

5 Syntaxanalyse

Die Syntaxanalyse erkennt die Struktur des Eingabewortes. Der entsprechende Teil des Compilers heißt „Parser“ (engl. *to parse* für „Zergliedern“ oder „grammatikalisch Bestimmen“). Als Eingabe erhält der Parser den Token-Stream vom Scanner aus der vorhergehenden lexikalischen Analyse. Die Ausgabe besteht aus einem sogenannten „Parse-Baum“ oder einem „Abstract Syntax Tree“ (AST), aus denen in der Synthesephase letztlich das Ausgabewort in der Zielsprache erzeugt werden kann.

5.1 Kontextfreie Grammatiken

Jede Programmiersprache wird durch eine Menge von syntaktischen Regeln beschrieben, die festlegen, welche Eingabewörter syntaktisch wohlgeformt sind. Beispielsweise könnte ein Funktionsaufruf wie folgt definiert sein:

> *„Ein Bezeichner gefolgt von einer in ()-geklammerten, beliebig langen Liste von komma-getrennten Ausdrücken.“*

Da bspw. ein Ausdruck wiederum geklammert werden darf, reichen die vorgestellten regulären Sprachen nicht mehr aus, diese zu beschreiben. Sie sind nicht mächtig genug, um verschachtelte Konstrukte, wie den geklammerten Ausdrücken oder verschachtelten Schleifen in Programmiersprachen zu erkennen. Ein Beweis dafür lässt sich mit Hilfe des sogenannten „Pumping Lemma“ erbringen, auf den an dieser Stelle verzichtet wird. Für die Beschreibung syntaktischer Konstrukte wird hier der nächste Typ von Grammatik in der Chomsky-Hierarchie, die Typ-2-Grammatiken, die als „kontextfreie Grammatiken“ bezeichnet werden, vorgestellt. Sie sind das Mittel zum formalen Beschreiben der Syntax einer Programmiersprache.

Definition Eine *kontextfreie Grammatik* G ist ein Quadtrupel $G = (N, T, P, S)$ mit [HU00]

- einer Menge von Nichtterminalen N
- einer Menge von Terminalen T wobei $N \cap T = \emptyset$ gelten muss
- einer Menge von Produktionen P mit der Form $A \rightarrow \alpha$ wobei A ein Nichtterminal ist mit $A \in N$ und α eine Satzform mit $\alpha \in (N \cup T)^*$.
- einem Startsymbol S mit $S \in N$

1. Terminale sind die Grundsymbole, aus denen ein Quelltext bestehen kann. In diesem Kontext sind Symbole bzw. Tokens, wie Schlüsselwörter, Bezeichner und Literale, synonym als Terminale zu verstehen.

2. Nichtterminale werden oft auch als „Variablen“ oder im Zusammenhang mit den natürlichen Sprachen als „synktaktische Kategorien“ bezeichnet. Sie stellen jeweils eine eigene Sprache dar (ähnlich wie die Definitionen aus einer regulären Definition). Der vorgestellte Funktionsaufruf, aber auch eine Schleife, eine bedingte Anweisungen usw. einer Programmiersprache sind Nichtterminale.

3. Der Aufbau der Nichtterminale aus den Terminalem und Nichtterminalen wird durch die Menge der Produktionen beschrieben. Auf der linken Seite einer Produktion steht stets ein Nichtterminal und auf der rechten Seite eine Aneinanderreihung von Nichtterminalen und Terminalen, die als „Satzform“ bezeichnet werden.

4. Das Startsymbol ist ein ausgezeichnetes Nichtterminal. Von ihm aus beginnt die sogenannte „Ableitung“ für ein Eingabewort.

Beispiel Folgende Grammatik beschreibt eine Sprache in der geklammerte Ausdrücke von binären Zahlen geschrieben werden können: $G = (\{E, A\}, \{+, -, 0, 1, (,)\}, P, E)$ mit $P =$

$$
\begin{aligned}
&E \to (E + E)\\
&E \to (E - E)\\
&E \to A\\
&A \to AA\\
&A \to 0\\
&A \to 1
\end{aligned}
$$

Wenn $A \to \alpha_1, A \to \alpha_2, \ldots, A \to \alpha_k$ die Produktionen für das Nichtterminal A sind, dann kann auch verkürzt

$$A \to \alpha_1 \mid \alpha_2 \mid \ \ldots \ \mid \alpha_k$$

geschrieben werden. Die | sind als „oder“ zu lesen. Für E aus dem oberen Beispiel könnte also verkürzt auch

$$E \to (E + E) \mid (E - E) \mid A$$

geschrieben werden.

Eine kontextfreie Grammatik, im Folgenden nur noch „Grammatik“ genannt, beschreibt eine kontextfreie Sprachen, formal als $L(G)$ geschrieben, mit G als Grammatik. Um zu beschreiben, welche Wörter Element dieser Sprache sind, muss zunächst die *Ableitungsrelation* eingeführt werden.

Definition Die *Ableitungsrelation* $\Rightarrow_G \subseteq (N \cup T)^*$ einer Grammatik $G = (N, T, P, S)$ ist definiert durch [HU00]:

$$\alpha A \gamma \ \Rightarrow_G \ \alpha\beta\gamma,$$

wenn $A \rightarrow \beta \in P$ und $\alpha, \gamma \in (N \cup T)^*$ gilt.

Man sagt, dass die Produktion $A \rightarrow \beta$ auf die Satzform $\alpha A \gamma$ angewendet wird, um $\alpha\beta\gamma$ zu erhalten, oder anders ausgedrückt: $\alpha\beta\gamma$ kann *direkt* aus $\alpha A \gamma$ abgeleitet werden. Da die Produktion $A \rightarrow \beta$ immer auf $\alpha A \gamma$ angewandt werden kann, unabhängig vom Linkskontext α und Rechtskontext γ, wird die Grammatik als *kontextfrei* bezeichnet. Auch die Namensgebung für *Terminale* bzw. *Nichtterminale* wird ersichtlich, wenn man bedenkt, dass aus einer Satzform, die ausschließlich aus Terminalen besteht, nicht abgeleitet werden kann, die Ableitung somit endet bzw. *terminiert.*

Seien $\alpha_1, \alpha_2, \ldots, \alpha_k$ mit $k \geq 1$ Satzformen aus $(N \cup T)^*$ und es gelte:

$$\alpha_1 \ \Rightarrow_G \ \alpha_2, \alpha_2 \ \Rightarrow_G \ \alpha_3, \ \ldots, \ \alpha_{k-1} \ \Rightarrow_G \ \alpha_m,$$

dann kann man verkürzt $\alpha_1 \ \Rightarrow_G^* \ \alpha_m$ schreiben. Man sagt dazu, dass α_m in Grammatik G aus α_1 *ableitbar* ist. Das bedeutet, dass $\Rightarrow_G^*$ die reflexive und transitive Hülle von $\Rightarrow_G$ ist. Desweiteren kann man mit $\alpha \Rightarrow_G^n$ ausdrücken, dass α in genau n Schritten aus β ableitbar ist. Wenn aus dem Kontext hervorgeht, um welche Grammatik es sich handelt, wird i. a. der Name der Grammatik am Relationszeichen weggelassen. Statt $\Rightarrow_G$ wird $\Rightarrow$, statt $\Rightarrow_G^*$ wird $\Rightarrow^*$ und statt $\Rightarrow_G^n$ wird $\Rightarrow^n$ geschrieben.

Die von einer Grammatik $G = (N, T, P, S)$ erzeugten Sprache $L(G)$ kann nun beschrieben werden durch:

$$L(G) = \{w \in T^* \mid S \ \Rightarrow_G^* \ w\}$$

Das sind somit alle Wörter über den Terminalen, die in der Grammatik aus dem Startsymbol ableitbar sind. Zwei Grammatiken G_1 und G_2 sind äquivalent, wenn $L(G_1) = L(G_2)$ gilt.

Beispiel Gegeben sei die Grammatik $G = (N, T, P, S)$ mit $N = \{S\}$, $T = \{a, b\}$ und $P = \{S \rightarrow aSb, S \rightarrow ab\}$. Wendet man die erste Produktion $S \rightarrow aSb$ (n-1)-mal und anschließend die zweite Produktion $S \rightarrow ab$ einmal an, ergibt dies:

$$S \ \Rightarrow \ aSb \ \Rightarrow \ aaSbb \ \Rightarrow \ a^3Sb^3 \ \Rightarrow \ \ldots \ \Rightarrow \ a^{n-1}Sb^{n-1} \ \Rightarrow \ a^nb^n.$$

Jedes Mal, wenn die erste Produktion angewandt wird, bleibt die Anzahl für a und b gleich. Für die zweite Produktion gilt das offensichtlich auch. Die von G erzeugte Sprache ist damit $L(G) = \{a^nb^n \mid n \geq 1\}$. Damit wäre auch gezeigt, dass Klammerausdrücke durch kontextfreie Grammatiken beschrieben werden können, wenn man statt a die öffnende Klammer (und für b die schließende Klammer) einsetzt.

5.2 Ableitungsbäume

Bei den EAs konnte man durch Angabe des dazugehörigen Pfades zeigen, wie der EA ein gegebenes Eingabewort abarbeitet, wobei die Markierungen der Kanten entlang des Pfades das Wort ergeben. Für die Syntaxanalyse gibt es eine ähnliche Möglichkeit, mit Hilfe dieser man die Struktur eines Eingabewortes darstellen kann – den „Ableitungsbaum“, oft auch „Syntaxbaum“ oder engl. „Parse-Tree“ genannt.

Definition Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Ein Baum ist ein *Ableitungsbaum*, wenn gilt[HU00]:

1. Jeder Knoten ist mit einem Zeichen aus $N \cup T \cup \{\epsilon\}$ markiert.
2. Die Wurzel ist mit S markiert.
3. Wenn ein innerer Knoten mit A markiert ist, dann muss $A \in N$ sein.
4. Wenn n mit A markiert ist und die Knoten $n_1, n_2, \ldots, n_k$ dessen Söhne und von links her mit $X_1, X_2, \ldots, X_k$ markiert sind, dann muss

   $$A \rightarrow X_1, X_2, \ldots, X_k$$

   eine Produktion aus P sein.
5. Wenn der Knoten n mit ϵ markiert ist, dann ist n ein Blatt und der einzige Sohn seines Vaters.

An den Blättern des Baumes lässt sich von links nach rechts das Eingabewort bestehend aus Terminalen ablesen (die ϵ werden dabei überlesen), daher spielt die Reihenfolge der Kinder eines Knotens eine Rolle.

Beispiel Sei $G = (N, T, P, S)$ eine Grammatik mit $N = \{E, A\}$, $T = \{+, -, 0, 1, (,)\}$, $P = E \rightarrow (E + E) \mid (E - E) \mid A,\ A \rightarrow AA \mid 0 \mid 1$ und sei folgende Ableitung gegeben:

$$E \Rightarrow (E+E) \Rightarrow (A+E) \Rightarrow (AA+E) \Rightarrow (0A+E) \Rightarrow (01+E) \Rightarrow (01+A) \Rightarrow (01+1)$$

Der dazugehörige Ableitungsbaum ist in Abbildung 14 dargestellt.

Die Beziehungen zwischen *Ableitung*, *Ableitungsbaum* und *erzeugtem Wort* sind nicht eindeutig. Es gilt:

1. Ein Ableitungsbaum kann mehrere Ableitungen darstellen. Habe bspw. eine Grammatik $P = S \rightarrow AB, A \rightarrow a, B \rightarrow b$. Die Ableitungen sind

   $S \Rightarrow AB \Rightarrow aB \Rightarrow ab$ und

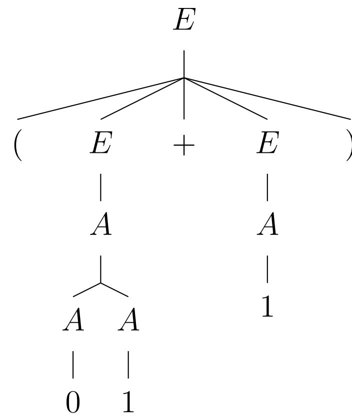


Abbildung 14: Ableitungsbaum für $(01+1)$

$$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab,$$

je nachdem, ob zuerst A oder B ersetzt wird. Aber der Ableitungsbaum in Abbildung 15 ist für beide Ableitungen gleich.

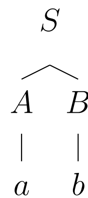


Abbildung 15: Ableitungsbaum für $S \Rightarrow^* ab$

2. Eine Ableitung kann durch mehrere Ableitungsbäume dargestellt werden. Habe beispielsweise eine Grammatik $P = S \rightarrow AA, A \rightarrow AA \mid a$. Für die Ableitung

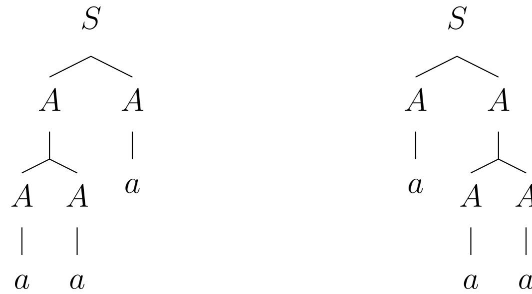
   $$S \Rightarrow AA \Rightarrow AAA \Rightarrow^3 aaa$$

   existieren die zwei Ableitungsbäume aus Abbildung 16a und 16b je nachdem, ob das erste oder zweite Nichtterminal im zweiten Ableitungsschritt ersetzt wird.

3. Ein Wort kann durch mehrere Ableitungen erzeugt werden. Für die Grammatik mit $P = S \rightarrow AB, A \rightarrow a, B \rightarrow b$ aus den oberen Beispiel kann man bereits erkennen, dass das Wort ab durch

   $$S \Rightarrow AB \Rightarrow aB \Rightarrow ab \text{ und}$$
   $$S \Rightarrow AB \Rightarrow Ab \Rightarrow ab,$$

   erzeugt wird.



(a) erstes Nichtterminal wurde ersetzt (b) zweites Nichtterminal wurde ersetzt

Abbildung 16: Ableitungsbäume für $S \Rightarrow AA \Rightarrow AAA \Rightarrow^3 aaa$

4. Ein Wort kann mehrere Ableitungsbäume besitzen. Auch das wurde bereits gezeigt, in dem für das Wort *aaa* die Ableitungsbäume aus Abbildung 16a und 16b konstruiert wurden.

5.3 Links- und Rechtsableitung

Bisher war es egal, welches Nichtterminal bei einem Ableitungsschritt ersetzt wurde. So kann, wie oben gezeigt wurde, ein Wort mehrere Ableitungsbäume erzeugen. Für eine Programmiersprache ist es jedoch wichtig, für ein Eingabewort einen *eindeutigen* Ableitungsbaum zu erhalten. Man muss sich somit auf eine der beiden Ableitungen „Linksableitung“ oder „Rechtsableitung“ einigen, die für unterschiedliche Parsertypen geeignet sind. Sei

$\alpha A\gamma \Rightarrow \alpha\beta\gamma$ ein Ableitungsschritt mit $A \to \beta \in P$ und $\alpha, \gamma \in (N \cup T)^*$.

Enthält α ausschließlich null oder mehr Terminale, wird der Ableitungsschritt als „Linksableitung“ bezeichnet – geschrieben als $\alpha A\gamma \Rightarrow_l \alpha\beta\gamma$. Analog dazu: Enthält γ ausschließlich null oder mehr Terminale, dann handelt es sich um eine „Rechtsableitung“ – geschrieben als $\alpha A\gamma \Rightarrow_r \alpha\beta\gamma$. Die Festlegung auf Links- oder Rechtsableitung ändert nichts an der erzeugten Sprache. Es gilt stets:

$$L(G) = \{w \in T^* \mid S \Rightarrow_l^* w\} = \{w \in T^* \mid S \Rightarrow_r^* w\}$$

5.4 LL(k)-Grammatiken

Es gibt universelle Parsing-Methoden wie z. B. der Cocke-Younger-Kasami-Algorithmus (CYK-Algorithmus), die für alle kontextfreien Grammatiken funktionieren. Allerdings muss hier die Grammatik zunächst in die sogenannte „Chomsky-Normalform“ überführt werden, was den Ableitungsbaum am Ende unübersichtlicher macht und weiterhin ist

die kubische Zeitkomplexität ungeeignet. In [ASU99] werden solche Methoden für den praktischen Einsatz als „zu ineffizient“ bezeichnet.

Stattdessen werden in der Praxis die *Top-Down-* und *Bottom-Up-Methoden* eingesetzt [ASU99]. Erstere konstruiert den Ableitungsbaum von der Wurzel aus, also dem Startsymbol der Grammatik, nach unten zu den Blättern (Terminalen). Die Zweite arbeitet genau andersherum und konstruiert den Ableitungsbaum von den Blättern hin zur Wurzel. Beide Methoden funktionieren allerdings nicht mit jeder kontextfreien Grammatik. Sie sind auf Teilklassen von Grammatiken beschränkt, wie die LL- und LR-Klassen. Für den später beschriebenen *Recursive-Descent-Parser* – ein Top-Down-Parser – werden nun die LL(k)-Grammatiken eingeführt, aus diesen sich solch ein Parser konstruieren lässt.

Damit die LL(k)-Grammatiken definiert werden können, muss zunächst die sogenannte $first_k$-Menge eingeführt werden.

Definition Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Die $first_k$-Menge einer Satzform $\alpha \in (N \cup T)^*$ mit $k \in \mathbb{N}_0$ ist definiert als:

$$\begin{aligned} first_k(\alpha) = \{v \in T^k \mid & \text{ es existiert ein } w \in T^*, \text{ so dass } \alpha \Rightarrow^* vw \text{ gilt}\} \\ \cup \{v \in T^{<k} \mid & \alpha \Rightarrow^* v\} \end{aligned}$$

Die $first_k$-Menge bestimmt für eine Satzform alle Wörter über den Terminalen mit der Länge $\leq k$, die Präfix von einem aus der Satzform ableitbaren Wort sind. Damit kann ein Parser durch Lesen eines Wortes der Länge k, also durch lesen von k Terminalen, bereits entscheiden, ob dieses Wort sich möglicherweise aus der Satzform ableiten lässt.

Beispiel Sei G eine Grammatik mit $G = (\{S\}, \{a, b\}, P, S)$ und $P = S \to aSb \mid \epsilon$, dann sind folgende $first_k$-Mengen für G:

$$\begin{aligned} first_1(ab) &= \{a\} \\ first_3(ab) = first_2(ab) &= \{ab\} \\ first_3(S) &= \{\epsilon, ab, aab, aaa\} \end{aligned}$$

Definition Sei $G = (N, T, P, S)$ und seien $A \to \beta$ und $A \to \gamma$ Produktionen aus P mit $\beta \neq \gamma$, dann ist G eine *LL(k)-Grammatik* wenn gilt:

$$first_k(\beta\alpha) \cap first_k(\gamma\alpha) = \emptyset \text{ mit } w \in T^* \text{ und } \alpha, \beta, \gamma \in (N \cup T)^*$$

Das besagt, dass man bei diesem Grammatiken durch Lesen von k Terminalen im Voraus immer entscheiden kann, welche Produktion als nächstes abgeleitet werden muss.

5.5 Elimination von Links-Rekursionen

Eine Grammatik ist links-rekursiv, wenn es für ein Nichtterminal A und eine Satzform α eine Ableitung $A \Rightarrow^+ A\alpha$ gibt [ASU99]. Da LL(k)-Parser stets eine Linksableitung durchführen, würde dies zu einer Endlosschleife führen. Es wird nun der Algorithmus *Elimination von Links-Rekursionen* vorgestellt, mit dem Links-Rekursionen beseitigt werden, ohne die Sprache zu ändern. Er stammt aus [ASU99].

Algorithmus Elimination von Links-Rekursionen

Eingabe: Eine Grammatik G ohne Zyklen (Ableitungen der Form $A \Rightarrow^+ A$) und ϵ-Produktionen(Produktionen der Form $A \rightarrow \epsilon$).

Ausgabe: Eine äquivalente Grammatik ohne Links-Rekursionen. Sie kann u. U. ϵ-Produktionen enthalten.

Methode:

1. Ordne die Nichtterminale in einer beliebigen Reihenfolge $A_1, A_2, \ldots, A_n$ an.

2. *for* i := 1 *to* n *do begin*
    for j := 1 *to* $i - 1$ *do begin*
        ersetze jede Produktion der Form $A_i \rightarrow A_j\gamma$
        durch die Produktionen $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k$,
        wobei $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ alle aktuellen A_j-Produktionen sind;
        eliminiere *direkte Links-Rekursionen* in den A_i-Produktionen
    end
  end

Die *direkte Elimination von Links-Rekursionen* ist einfacher zu beschreiben. Eine Produktion der Form $A \rightarrow A\alpha \mid \beta$ kann ersetzt werden durch

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

ohne die von der Grammatik erzeugten Sprache zu verändern.

Beispiel Sei eine Grammatik gegeben mit folgenden Produktionen

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid 0 \mid 1$$

Nach Anwenden der *direkten Elimination von Links-Rekursionen* auf E und T lauten die neuen Produktionsregeln:

$$E \to TE'$$
$$E' \to TE' \mid \epsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \epsilon$$
$$F \to (E) \mid 0 \mid 1$$

Beispiel Sei eine Grammatik gegeben mit folgenden Produktionen

$$A \to Aa \mid b$$
$$A \to Ac \mid Sd$$

Hier kann keine *direkte Elimination von Links-Rekursionen* angewandt werden, deswegen muss der Algorithmus zur *Elimination von Links-Rekursionen* angewandt werden:

1. Die Nichtterminale werden als $S,\ A$ geordnet
2. In S kommt keine direkte Links-Rekursion vor, deswegen kann zu $i = 2$ gesprungen werden. Für $i = 2$ wird die S-Produktion in $A \to Sd$ ersetzt zu:

   $$A \to Ac \mid Aad \mid bd$$

   Dann wird die direkte Links-Rekursion in A eliminiert.

Es ergibt sich folgende Menge an Produktionen:

$$S \to Aa \mid b$$
$$A \to bdA' \mid A'$$
$$A' \to cA' \mid adA'$$

5.6 Links-Faktorisierung

Gibt es in einer Grammatik für ein Nichtterminal mindestens zwei Produktionen der Form $A \to \alpha$ und $A \to \beta$, deren Ableitung sich nicht durch ein Terminal entscheiden lässt, also $first_1(\alpha) \cap first_1(\beta) \neq \emptyset$, dann kann ein LL(1)-Parser für die Grammatik nicht konstruiert werden. Mit Hilfe der *Links-Faktorisierung* wird versucht, diese Entscheidung soweit wie möglich heraus zu zögern [ASU99]:

Das Problem lautet allgemein: Für zwei Produktionen $A \to \alpha\beta_1$ und $A \to \alpha\beta_2$, ist nach Lesen eines aus α ableitbaren Wortes nicht entscheidbar, ob nach $\alpha\beta_1$ oder $\alpha\beta_2$ abzuleiten ist. Die Idee ist, die Entscheidung vorerst zurückzustellen und $\alpha A'$ abzuleiten. Durch Links-Faktorisierung ergeben sich die neuen Produktionen als:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Beispiel Sei eine Grammatik mit folgenden Produktionsregeln gegeben:

$$S \rightarrow if\ E\ then\ S\ else\ S\ endif$$
$$S \rightarrow if\ E\ then\ S\ endif$$

Hier ist $\alpha = if\ E\ then\ S$, $\beta_1 = else\ S\ endif$ und $\beta_2 = endif$. Nach Anwenden der Links-Faktorisierung ergibt sich:

$$S \rightarrow if\ E\ then\ SS'$$
$$S' \rightarrow else\ S\ endif \mid endif$$

Es ist zu erkennen, dass die beiden Produktionen für S' nun durch Lesen eines Terminals voneinander unterschieden werden können.

Sowohl die Elimination von Links-Rekursionen als auch die Links-Faktorisierung können jedoch nicht jede kontextfreie Grammatik in eine LL(k)-Grammatik umformen. Das lässt sich damit begründen, dass LL(k)-Grammatiken eine Untermenge von kontextfreien Grammatiken sind.

5.7 Entwurf eines Recursive-Descent-Parsers

Generell kann jede kontextfreie Sprache von einem sogenannten „nicht deterministischen Kellerautomaten“ erkannt werden. Schränkt man die Grammatik auf eine LL(k)-Grammatik ein, so genügt ein „deterministischer Kellerautomat“. Ein Kellerautomat ist ein endlischer Automat, der um einen Keller (Stack) erweitert wird. Für die Implementierung eines Parsers könnte somit ein solcher Kellerautomat simuliert werden. Bedenkt man jedoch, dass prozeduale Programmiersprachen ebenfalls auf Basis eines Stacks arbeiten, kann man auf die Simulation eines Kellerautomaten verzichten, und auf eine intuitivere Methode zurückgreifen.

Ein *Recursive-Descent-Parser* ist ein Top-Down-Parser, der nach der Methode des *rekursiven Abstiegs* (engl. Recursive Descent) arbeitet. Er ist für LL(k)-Grammatiken geeignet. In [Lan06] wird die Methode des rekursiven Abstiegs beschrieben:

1. Für jedes Nichtterminal wird eine gleichnamige Prozedur geschrieben.
2. In der Prozedur wird die rechte Seite aller Produktionen des Nichtterminals behandelt.

3. Nichtterminale werden durch Aufruf der gleichnamigen Prozedur behandelt, Terminale durch Abarbeiten des zugehörigen Tokens aus dem Eingabewort.

4. Gibt es mehrere Produktionen zu einem Nichtterminal, also mehrere Alternativen, muss auf Grund der im Voraus gelesenen k-Tokens, dem sogenannten „Lookahahead“ (engl. Vorgriff) – eine Fallunterscheidung durchgeführt werden.

Jeder Aufruf einer Prozedur für ein Nichtterminal entspricht einem Ableitungsschritt. Für ϵ-Produktionen wird zusätzlich die Konvention vereinbart, dass sie in einer Fallunterscheidung zuletzt geprüft wird.

Beispiel Seien folgende Produktionsregeln einer LL(1)-Grammatik gegeben:

$$S \to if\ E\ then\ SS' \mid a$$
$$S' \to else\ S\ endif \mid endif$$

Die $first_1$-Mengen sind:

$$first_1(if\ E\ then\ SS') = \{if\}$$
$$first_1(a) = \{a\}$$
$$first_1(else\ S\ endif) = \{else\}$$
$$first_1(endif) = \{endif\}$$

Alle $first_1$-Mengen für jede Produktion eines Nichtterminals sind disjunkt, deshalb reicht ein Lookahead von einem Terminal aus. Die Prozeduren für S und S' sind nach der Methode des rekursiven Abstiegs wie folgt aufgebaut:

```
void S() {
   if (token == if) {
      nextToken();
      E();
      match(then);
      S();
      S'();
   } else if (token = a) {
      nextToken();
   } else {
     error(if oder a erwartet);
   }
}

void S'() {
   if (token == else) {
      nextToken();
```

```
        S();
        match(endif);
    } else if (token = endif) {
        nextToken();
    } else {
        error(else oder endif erwartet);
    }
}

void match(t) {
    if (token == t)
        nextToken();
    else
        error(t erwartet);
}
```

Die Prozedur `nextToken` fordert vom Scanner das nächste Token an und legt es in `token` ab. Damit ist `token` der aktuelle Lookahead.

Manche rekursiven Aufrufe lassen sich durch Iteration ersetzen. Zum Beispiel bei folgenden Produktionen:

$$S \rightarrow aS \mid bS \mid \epsilon.$$

Hier liegt eine *endständige Rekursion* vor, da sich S am Ende für aS und bS selbst aufruft. Das kann durch Iteration ersetzt werden:

```
void S() {
    while (true) {
        if (token == a) {
            nextToken();
        } else if (token == b) {
            nextToken();
        } else {  // Epsilon-Produktionen
            break; // werden zu letzt "geprueft"
        }
    }
}
```

Abschließend sei auf Parser-Generatoren wie Yacc (Yet another Compiler Compiler) oder ANTLR (Another Tool for Language Recognition) hingewiesen, die aus einer gegebenen Grammatik-Definition einen effektiven Parser generieren. Beispielsweise generiert ANTLR LL(k)-Parser für LL(k)-Grammatiken für ein beliebiges k.

6 Semantische Analyse und Synthesephase

Bislang konnte ein Parser lediglich eine Antwort darauf geben, ob ein Eingabewort Teil einer Sprache ist. In diesem Abschnitt werden die kontextfreien Grammatiken um Attribute und semantische Regeln zu den *attributierten Grammatiken* ergänzt, um einerseits zur Kompilierzeit Regeln wie „Wurde die Funktion mit der richtigen Anzahl an Parametern aufgerufen?“ zu prüfen, andererseits auch eine Ausgabe zu erzeugen. Sie wurden 1968 von Knuth eingeführt.

Definition Sei $G = (N, T, P, S)$ eine kontextfreie Grammatik.

1. Jedem Grammatiksymbol $X \in (N \cup T)$ ist eine endliche Menge an Attributen $A(X)$ zugeordnet. Das Attribut $a \in A(X)$ wird auch mit $X.a$ bezeichnet. Es gilt $A(X) \cap A(Y) \neq \emptyset \Rightarrow X = Y$.

2. Die Menge der Attribute $A(X)$ ist disjunkt zerlegt in:

   - die Menge der *ererbten Attribute* (engl. inherited) $A_I(X)$ und
   - die Menge der synthetisierten Attribute $A_S(X)$,

   d. h. es gilt $A(X) = A_I(X) \cup A_S(X)$ und $A_I(X) \cap A_S(X) = \emptyset$.

3. Jeder Produktion $X \to Y_1 \dots Y_k \in P$ mit $X \in N$, $Y_i \in (N \cup T)$ und $1 \leq i \leq k, k \geq 0$ ist eine endliche Menge von semantischen Regeln.

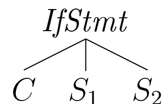
Eine *attributierte Grammatik* (oft auch Attributgrammatik genannt), ist eine kontextfreie Grammatik G mit Attributierung der Nichtterminalen und Terminalen und mit einer Zuordnung semantischer Regeln zu allen Produktionen.

Es ist nicht vorgeschrieben, was ein Attribut darstellt, bspw. eine Speicheradresse, einen Typ, einen ausgewerteten Ausdruck usw. Der Wert eines *synthetisierten Attributs* an einem Knoten eines Ableitungsbaums errechnet sich aus den Werten der Attribute der Nachfolger; der Wert eines *ererbten Attributs* wird aus den Werten der Geschwister und des Vorgängers berechnet [ASU99]. Die semantischen Regeln geben die Berechnung der Attribute an, können aber auch mit Seiteneffekten, wie dem Aufbau eines abstrakten Syntaxbaums, einem Eintrag in eine Symboltabelle oder der Ausgabe einer Fehlermeldung, verbunden sein.

Im Folgenden soll die attributierte Grammatik dazu genutzt werden, einen abstrakten Syntaxbaum aufzubauen.

6.1 Abstrakte Syntaxbäume

Bei der Syntaxanalyse wird implizit ein Ableitungsbaum aufgebaut (beim vorgestellten Recursive-Descent-Parser durch den rekursiven Prozeduraufruf), unabhängig davon, ob er explizit gespeichert wird. Einige Informationen werden jedoch für die weitere semantische Analyse bzw. Synthesephase nicht benötigt. Wurde bspw. einmal eine bedingte Anweisung erkannt, kann darauf verzichtet werden, die Schlüsselworte `if`, `then`, `else` u. a. zu speichern.



Ein abstrakter Syntaxbaum, engl. *Abstract Syntax Tree* (AST), ist eine verdichtete Form eines Ableitungsbaums, die zur Darstellung von Sprachkonstrukten hilfreich ist. In einem AST kommen Operatoren und Schlüsselworte nicht als Blätter vor, sondern sie sind mit den inneren Knoten assoziiert, die Vorgänger solcher Blätter im Ableitungsbaum sind. Zudem ist eine häufig vorkommende Vereinfachung, dass Ketten von Kettenproduktionen (`123` ist ein Literal, ist ein Faktor, ist ein Term, ist ein Ausdruck) in sich zusammenfallen (es bleibt `123` als Operand stehen) [ASU99].

Um ein AST aufzubauen, müssen in den semantischen Regeln die Knoten des Baums aufgebaut werden. Das folgende Beispiel soll das Prinzip anhand einer attributierten Grammatik für Ausdrücke verdeutlichen.

Beispiel Gegeben sei eine attributierte Grammatik mit folgenden Produktionen und semantischen Regeln. Das Beispiel stammt aus [ASU99].

Produktion	Semantikregel
$E \to E_1 + T$	$E.n := mknode(+, E_1.n, T.n)$
$E \to E_1 - T$	$E.n := mknode(-, E_1.n, T.n)$
$E \to T$	$E.n := T.n$
$T \to (E)$	$T.n := E.n$
$T \to id$	$T.n := mkleaf(id, id.entry)$
$T \to num$	$T.n := mkleaf(num, num.val)$

Dabei erstellen die Funktionen `mknode(op, left, right)` einen Knoten und `mkleaf(id, entry)` bzw. `mkleaf(num, val)` Blätter eines ASTs und geben die Referenz darauf zurück. Die Attribute $E.n$ und $T.n$ speichern die Referenz zum zugehörigen Knoten bzw. Blatt. Tritt ein Nichtterminal sowohl auf der linken als auch rechten Seite einer Produktion auf, so ist es üblich wie bei $E \to E_1 + T$, das Nichtterminal auf der rechten Seite von links nach rechts ab 1 über einen Index durchzunummerieren.

Für den Ausdruck $a - 4 + c$ ist der mit den Semantikregeln erstellte AST in Abbildung 17 angegeben. Man kann erkennen, dass die Kettenproduktionen zusammengefallen sind.

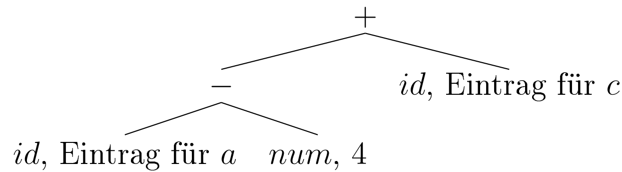


Abbildung 17: Konstruierter AST für $a - 4 + c$

Das Erstellen eines ASTs ist nicht unbedingt notwendig; die Übersetzung kann auch durch die Semantikregeln direkt erfolgen (interessant für sogenannte „Ein-Pass-Compiler", die in einem Lauf übersetzen). Für die erste Optimierung in der Synthesephase ist jedoch ein AST von Vorteil, weil Knoten geändert, gelöscht und hinzugefügt werden können.

6.2 Optimierung des abstrakten Syntaxbaums

Die Optimierung ist ein umfangreiches Gebiet im Compilerbau, deshalb wird hier nur ein Beispiel gegeben, um einen Eindruck davon gewinnen zu können.

Beispiel Das *Constant Folding* ist ein Optimierungsverfahren, welches in vielen Compilern Anwendung findet. Es versucht Ausdrücke zur Kompilierzeit soweit wie möglich auszuwerten. Für die vorige Grammatik für Ausdrücke könnte eine Funktion $visit(Knoten\ n)$ für ein sehr einfaches Constant Folding wie folgt lauten:

```
if not isLeaf(n) then begin
    visit(n.left);
    visit(n.right);
    if isNum(n.left) and isNum(n.right) then begin
        case n.op of
            + : n := mkleaf(num, n.left.val + n.right.val);
            − : n := mkleaf(num, n.left.val − n.right.val);
        end
    endif
end
```

Von unten nach oben werden Operatorknoten im AST für Addition und Subtraktion, bei denen der linke und rechte Operand eine *num* (eine Ganzzahl) ist, ersetzt durch ein Blatt mit dem berechneten Wert aus den beiden Operanden. Für den Ausdruck $a + (1 + 2) - 3$ ist der AST in Abbildung 18 und der optimierte AST nach Anwenden der Funktion *visit* auf die Wurzel in Abbildung 19 dargestellt. Der Ausdruck konnte nur auf $a + 3 - 3$ und

nicht auf $a + 0$ verkürzt werden, da dieser Algorithmus nur jeweils eine Ebene des ASTs betrachtet.

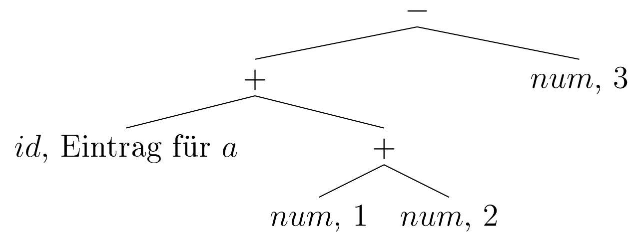


Abbildung 18: Unoptimierter AST für $a + (1 + 2) - 3$

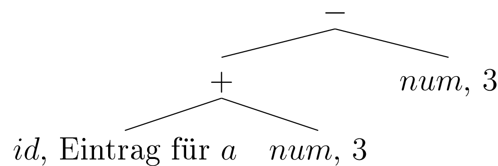


Abbildung 19: Optimierter AST für $a + (1 + 2) - 3$

6.3 Code-Generierung

Abhängig von der Zielsprache des Compilers gibt es viele Möglichkeiten das Ausgabewort zu erzeugen: Sei es Maschinen-Code für eine bestimmte Prozessor-Architektur, eine andere Programmiersprache, wie C, oder Byte-Code für eine virtuelle Maschine. Sowohl die Java-Plattform als auch die .NET-Plattform arbeiten nach letztem Prinzip. Sie erzeugen unabhängig von der Quellsprache Byte-Code (Java-Plattform) bzw. IL-Code (Intermediate Langugage, .NET-Plattform). Beide Plattformen arbeiten nach dem Prinzip einer Stack-Maschine. Dabei werden die Operanden auf den Stapel abgelegt und nach Aufruf des Operators durch das Ergebnis ersetzt. Bei der Java-Plattform wird diese Stack-Maschine zur Laufzeit auf der jeweiligen physischen Plattform ausgeführt. Besonders häufig ausgeführter Code kann durch die sogenannte „HotSpot-Technik“ erkannt werden und durch Maschinen-Code für die Zielplattform ersetzt werden. Beide Plattformen stellen unabhängig von der Quellsprache ein großes Framework an Funktionalität für den Zugriff auf Datenbanken, Lesen und Schreiben in Dateien, Arbeiten mit Benutzeroberflächen, Client-Server-Kommunikation u. v. a. zur Verfügung. Das entbindet den Entwickler eines Compilers für diese Plattformen davon, extra Bibliotheken für diese Funktionalitäten entwickeln zu müssen. So entstehen derzeit viele Programmiersprachen für die Java-Plattform wie Scala, JRuby und Clojure.

Anhand der Java-Plattform soll die Code-Erzeugung für typische Programmierkonstrukte in imperativen Sprachen aufgezeigt werden. Eine entsprechende Bibliothek zum Erzeu-

gen und Manipulieren von Java-Byte-Code ist das von der Apache Software Foundation stammende BCEL (Byte Code Engineering Library).

Beispiel Übersetzen von Ausdrücken
Für die obere attributierte Grammatik für Ausdrücke, die einen AST aus Operator-Knoten und Operanden-Blättern erzeugt hat, kann jeder Knoten wie folgt übersetzt werden:

1. Für den Operator-Knoten $(+, left, right)$: Übersetze $left$, Übersetze $right$ und füge `IADD` (Integer Addition) hinzu.

2. Für den Operator-Knoten $(-, left, right)$: Übersetze $left$, Übersetze $right$ und füge `ISUB` (Integer Subtraction) hinzu.

3. Für das *num*-Blatt: Füge `BIPUSH num.val` (Push Byte-Konstante als Integerwert) hinzu.

4. Für das *id*-Blatt: Füge `ILOAD id.entry` (Integer Load aus lokaler Variable) hinzu.

Für den optimierten Ausdruck $a + 3 - 3$, der als AST in Abbildung 19 dargestellt ist, würde der so erzeugte Byte-Code wie folgt aussehen:

```
ILOAD  1 // die lokale Variable a hat den Index 1
BIPUSH 3
IADD
BIPUSH 3
ISUB
```

Der Aufbau des Stacks nach abarbeiten der einzelnen Befehle ist in Abbildung 20 dargestellt.

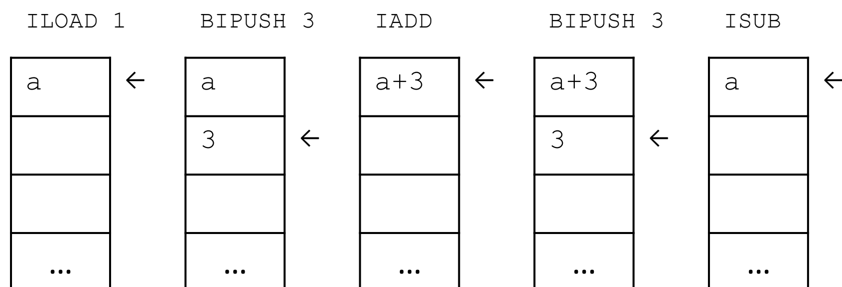


Abbildung 20: Aufbau des Stacks für $a + 3 - 3$

Beispiel Übersetzen von Konsolenausgaben
Eine Konsolenausgabe $print(E)$ kann durch

```
GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
Uebersetze E
INVOKEVIRTUAL java/io/PrintStream.println(I)V
```

übersetzt werden. Beispielsweise würde `print(a+3-3)` übersetzt werden zu:

```
GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
ILOAD  1
BIPUSH 3
IADD
BIPUSH 3
ISUB
INVOKEVIRTUAL java/io/PrintStream.println(I)V
```

Beispiel Übersetzen von Zählschleifen
Eine Zählschleife $for\ id = E_1\ to\ E_2\ do\ S\ next$ kann übersetzt werden zu:

```
Lk
  Uebersetze E1   // Push Startwert
  ISTORE id.entry // Pop in Zaehlvariable
  GOTO Lk+1
Lk+2
  Uebersetze S    // Schleifenkoerper
Lk+3
  IINC id.entry 1 // Zaehlvariable++
Lk+1
  ILOAD id.entry  // Push Zaehlvariable
  Uebersetze E2   // Push Endwert
  IF_ICMPLE Lk+2  // Pop a, Pop b, springe zu Lk+2 wenn a <= b
```

So würde der Quelltext

```
for i = 1 to 10 do
   print(i)
next
```

übersetzt werden zu:

```
L1 // Startwert
  BIPUSH 1
  ISTORE 1
  GOTO L2
L3 // Schleifenkoerper
  GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
  ILOAD 1
  INVOKEVIRTUAL java/io/PrintStream.println(I)V
L4 // Schritt
  IINC 1 1
L2 // Bedingung
```

```
    ILOAD 1
    BIPUSH 10
    IF_ICMPLE L3
```

Beispiel Übersetzen von bedingten Anweisungen
Eine bedingte Anweisung *if E then* S_1 *else* S_2 *endif* kann übersetzt werden zu:

```
Lk
  Uebersetze E
  IFEQ Lk+2 // Pop a, springe zu Lk+2 wenn a == 0
Lk+1
  Uebersetze S1
  GOTO Lk+3
Lk+2
  Uebersetze S2
Lk+3
```

So würde der Quelltext

```
if a then
   print(1)
else
   print(2)
endif
```

übersetzt werden zu:

```
L1 // a != 0?
  ILOAD 1
  IFEQ L3
L2 // print(1)
  GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
  BIPUSH 1
  INVOKEVIRTUAL java/io/PrintStream.println(I)V
  GOTO L4
L3 // print(2)
  GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
  BIPUSH 2
  INVOKEVIRTUAL java/io/PrintStream.println(I)V
L4
```

6.4 Peephole-Optimization

Nach der Code-Generierung erfolgt ein weiterer Optimierungsschritt über den generierten Code. Für Maschinensprache oder Byte-Code als Zielsprache findet oft eine sogenannte „Peephole-Optimization“ (engl. für Guckloch) statt. Dabei wird nur ein kleiner Ausschnitt (Guckloch) aus der erzeugten Befehlsfolge betrachtet. Es kommen hier Algorithmen zur Mustererkennung zum Einsatz. Das oben erwähnte BCEL unterstützt bspw. die Suche mit Hilfe von regulären Ausdrücken wie z. B. `NOP+(ILOAD|ALOAD)*`, das alle `NOP`-Anweisungen gefolgt von einer möglicherweise leeren Sequenz von `ILOAD`- oder `ALOAD`-Anweisungen im Byte-Code findet.
Auch die Peephole-Optimization ist ein sehr umfassender Bereich des Compilerbaus, weswegen nur ein typisches Beispiel gezeigt wird, um einen Eindruck davon gewinnen zu können.

Beispiel Optimieren von 2er-Potenz-Multiplikation durch bitweise Linksverschiebung
Multiplikation mit 2er Potenzen wie 2, 4, 8, 16 ... können durch bitweises Linksverschieben in einem Takt durchgeführt werden. Dabei gilt $x \cdot 2^n = x\ shl\ n$ mit $n \in \mathbb{N}_0$. Mit dem Ausdruck `~((k - 1) & k) == -1` kann in Java geprüft werden, ob k eine 2er-Potenz ist. Der Algorithmus für die Optimierung kann wie folgt lauten:

> Finde alle Befehlsfolgen `BIPUSH k`, `IMUL` mit `~((k - 1) & k) == -1`
> und ersetze sie durch `BIPUSH Sqrt(k)`, `ISHL`

So kann der generierte Byte-Code für `b = a*16 + 15`

```
ILOAD  1  // a
BIPUSH 16 //
IMUL
BIPUSH 15
IADD
ISTORE 2  // b
```

ersetzt werden zu

```
ILOAD  1  // a
BIPUSH 4  // = Sqrt(16)
ISHL      // Pop j, Pop k, Push j shl k
BIPUSH 15
IADD
ISTORE 2  // b
```

7 Zusammenfassung und Ausblick

Compilerbau gilt als eine der ältesten Disziplin der Informatik (etwa ab 1950 formalisiert betrachtet). Von den Anfängen als Kleene 1956 die regulären Ausdrücke eingeführt hatte, wenige Jahre später der erste Fortran-Compiler unter Leitung von Backus entwickelt wurde, 1965 der CYK-Parsing-Algorithmus für allgemeine kontextfreie Grammatiken von Younger, Cocke und Kasami vorgestellt wurde bis hin zu den Lexer- und Scanner-Generatoren 1975 von Lesk (Lex) und Johnson (Yacc). Der seit 1989 entwickelte Parser-Generator ANTLR von Parr ist heute ein viel eingesetztes Werkzeug für das Entwickeln von Programmiersprachen und dazugehörigen Compilern.
Diese und viele weitere Errungenschaften erlauben nur einen kleinen Einblick im Rahmen einer Studienarbeit in die vielen Teildisziplinen des Compilerbaus.

In dieser Studienarbeit wurden für die lexikalische Analyse die regulären Ausdrücke eingeführt, aus denen man mit der Thompson-Konstruktion endliche Automaten systematisch aufbauen kann und es wurde ein Algorithmus vorgestellt, um solch einen endlichen Automaten zu simulieren. Nicht betrachtet wurden Algorithmen, die diese endlichen Automaten bzgl. der Anzahl an Zuständen minimieren können.

Für die Syntaxanalyse wurden die kontextfreien Grammatiken vorgestellt. Beschränkt man diese auf LL(k)-Grammatiken, lassen sich daraus wie gezeigt, einfache Recursive-Descent-Parser implementieren. Er lässt sich auch nicht rekursiv mit einem Stack und einer Tabelle als „prädiktiver Parser“ implementieren. Daneben gibt eine Reihe weiterer Parsing-Techniken, wie dem „Shift-Reduce-Parsing“ für LR(k)-Grammatiken oder dem CYK-Algorithmus für allgemeine kontextfreie Grammatiken.

Die Syntaxanalyse kann aber nur auf strukturelle Korrektheit eines Quelltextes prüfen. Um auch Regeln für die statische Semantik zu formulieren, wurden die kontextfreien Grammatiken zu den attributierten Grammatiken erweitert, mit denen man Regeln und Attribute mit den Grammatiksymbolen verbindet. Sie wurden dazu genutzt, um abstrakte Syntaxbäume aufzubauen, um in einem nächsten Schritt eine Optimierung durchzuführen. Hier wurde beispielhaft das Constant Folding vorgestellt. Auf Basis des abstrakten Syntaxbaums wurden für verschiedene imperative Sprachkonstrukte die Code-Generierung in Java-Bytecode vorgestellt. Schließlich wurde ein Beispiel für die Peephole-Optimization gegeben, die einen abschließenden Optimierungsschritt darstellt.

Ein wichtiger Aspekt eines Compilers ist die Fehlerbehandlung, die hier nicht vorgestellt werden konnte. Dazu gibt es eine Reihe von verschiedenen Ansätzen wie der „panischen Recovery“ oder der „konstrukt-orientierten Recovery“. Es wurden zudem bei der Coder-Generierung nur Beispiele für imperative Sprachkonstrukte gegeben. Für andere Paradigmen, wie den funktionalen Sprachen oder den objekt-orientierten Sprachen, gibt es andere Techniken. Das wären Ansätze für weitere Studienarbeiten.

Literatur

[ASU99] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau Teil 1.* Oldenbourg, 1999. 2. Auflage.

[BW09] Federico Biancuzzi and Shane Warden. *Visionäre der Programmierung – Die Sprache und ihre Schöpfer.* O'Reilly, 2009.

[HU00] John E. Hopcroft and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Oldenbourg, 2000. 4. Auflage.

[Lan06] Hans Werner Lang. *Algorithmen in Java.* Oldenbourg, 2006. 2. Auflage.

[Wir96] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus.* Addison-Wesley, 1996.