Kapitel 3

Übersetzer (Compiler): Von der Programmiersprache zur Maschinensprache

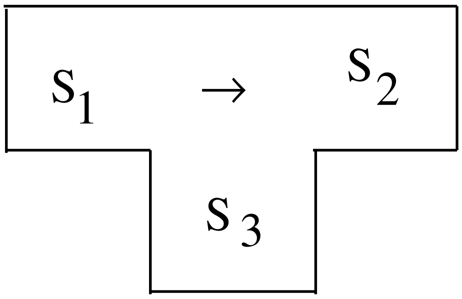
3.1 Compiler

Eine etwas vereinfachte Definition ist:

Ein Übersetzer (Compiler) ist ein Programm, das Sätze einer formalen Sprache S_1 in Sätze einer anderen formalen Sprache S_2 umwandelt .

z.B. Programme in einer Programmiersprache wie Haskell, Java oder PASCAL, in Programme einer anderen formalen Sprache, wie Byte-code oder die Maschinensprache eines Rechners.

Der Compiler selbst ist in einer Sprache S_3 geschrieben. Folgendes Puzzleteil kann man verwenden, um aufbauend auf lauffähigen Übersetzern weitere zu konstruieren, ohne neue Übersetzer schreiben zu müssen.



S_1, S_2, S_3 brauchen nicht notwendig verschieden voneinander zu sein.

Eine korrektere Definition ist:

> Ein Übersetzer (Compiler) ist ein Programm, das Sätze einer formalen Sprache S_1 in Sätze einer anderen formalen Sprache S_2 umwandelt, wobei die Semantik erhalten bleibt.

Typische Anwendungsfälle:

- Übersetzung eines Programms in einer Programmiersprache in ein Programm in einer Assemblersprache, so daß das Assemblerprogramm das richtige tut.

- Ein Compiler, der S_1-Programme in S_2-Programme übersetzt, damit ein weiterer Übersetzer von S_2 nach Assembler verwendet werden kann.

Die Syntax, d.h. die Beschreibung der möglichen (grammatikalisch korrekten) Quellprogramm ist i.a. definiert durch einen BNF (kontextfreie Grammatik). (s.u.)

Da die Semantik oft nur implizit oder nicht-formal spezifiziert ist, gibt es manchmal Fälle, in denen die Übersetzung nicht eindeutig definiert ist.

Üblich ist die Spezifikation bzw. Erwähnung einer "operationalen Semantik". D.h. die Angabe, welche Effekte bestimmte Anweisungen haben sollen.

Die wichtigsten Formen einer Semantik sind:

operationale Semantik: welchen Effekt haben bestimmte Anweisungen als Ausgabe oder im Hauptspeicher. Eine operationale Semantik ist äquivalent zum Programmieren eines Interpreters für alle zulässigen Programme einer Programmiersprache.

denotationale Semantik: Welche (mathematische) Bedeutung haben die Programmkonstrukte. Hier werden meist vollständige partielle Ordnungen verwendet, die es erlauben, z.B: einer While-Schleife ein Objekt als kleinste obere Schranke einer Folge zuzuweisen (Fixpunkt)

axiomatische Semantik: Welche (mathematische) Bedeutung haben die Programmkonstrukte. Verwendung der Prädikatenlogik. Zum Beispiel zur Spezifikation von Datentypen, oder auch axiomatische Beschreibung der operationalen Semantik (z.B. Axiome von Hoare)

transformationelle Semantik Wird oft verwendet um die Bedeutung komplexerer Anweisungen oder Ausdrücke einer Programmiersprache durch einen einfacheren Satz von Anweisungen zu erklären.

Höhere Programmiersprachen wie. Haskell, Java, Lisp, PASCAL, PROLOG, C, Ada, stellen Algorithmen in für den Menschen verständlicher Form dar. Diese sind gut strukturierbar und oft portabel, oder bis auf lokalisierbare Aufrufe portabel.

Eine Assemblersprache ist maschinenspezifisch (prozessorspezifisch). I.a. gibt es eine $1:1$-Zuordnung zwischen Assembler- und Maschinenbefehl. Es werden Symbole für Befehle, Adressen, Registernamen, Marken, Adressiermodi usw. benutzt. Die typischen Befehle und die operationale Semantik des Prozessors werden ausgenutzt.

Das Resultat der Assemblierung ist eine Folge codierter Maschinenbefehle, d.h. eine Folge von binären Speicherworten, die direkt vom Prozessor verstanden werden (d.h. ausgeführt werden können). Allerdings ist diese Folge für einen Prozessor andere Bauart nicht verständlich, bzw. bedeutet etwas anderes.

Auch auf dieser Ebene gilt: kennt man die operationale Semantik des Prozessors, dann kann man einen Interpreter (auf einem anderen Prozessor implementieren), der diesen Maschinenkode korrekt ausführt (allerdings indirekt, bzw. interpretiert).

Neben der Kompilierung höherer Programmiersprachen gibt es viele andere Anwendungen, z.B. die Übersetzung von

- Textverarbeitungssprachen, z.B. die Sprache von TEX
- Datenaustauschformaten, z.B. EDIF (Electronic Data Interchange Format)
- Hardwarebeschreibungssprachen, z.B. VHDL (VHSIC (Very High-Speed Integrated Circuit) Hardware Description Language)
- textuelle Analyse von Kommandosprachen
- usw.

3.2 Phasen eines Compilers

Lexikalische Analyse (scanning): ein Programm wird als Zeichenkette linear abgearbeitet und in Symbole (tokens) aufgeteilt

Z.B. "IF 13 == X1 THEN A " wird zu: ('IF', 13, ==, 'X1', 'THEN','A')

Syntaxanalyse(parsing): Symbole werden zu grammatikalischen Sätzen zusammengefaßt

Z.B. ('IF', 13, ==, 1, 'THEN', 'A') wird zu: ('IF' (13 == 'X1') 'THEN' 'A')

Semantische Analyse: Untersuchung auf semantische Fehler (z.B. Typüberprüfungen) Z.B. $1 +' a'$ wird zurückgewiesen.

Zwischencode-Erzeugung: Generierung von Code für eine abstrakte Maschine

Codeoptimierung: Verbesserung des Zwischencodes

Code-Erzeugung: Erzeugung von Assemblercode für eine reale Maschine

Wichtig für alle Phasen:

die Symboltabelle eine Datenstruktur, in der die wesentlichen Attribute (Typ, Gültigkeitsbereich, etc.) der Bezeichner eines Quellprogramms gesammelt werden

die Fehlerbehandlung die Fehler analysiert, dem Benutzer meldet und die Fortsetzung der Übersetzung ermöglichen soll.

Betrachtet man ein vollständiges System zum Übersetzen einer Programmiersprache, so kann es weitere Schritte geben:

Präprozessor Das Quellprogramm wird anhand einer Makrobibliothek oder anhand anderer Algorithmen verändert,(i.a. expandiert).

Übersetzer Dies ist der Aufruf des eigentlichen Compilers, der eine Assemblerausgabe erzeugt.

Assembler übersetzt das Assemblerprogramm in verschiebbarem Maschinenkode.

Lader/Binder Umsetzung in ausführbaren Maschinenkode plus Hinzunahme von anderem verschiebbarem Maschinenkode, z.B. aus Bibliothek

3.2.1 Trennung Front-End/Back-End

Um einen Compiler für mehrere Maschinen portierbar zu machen, ist eine Aufteilung in Front-End und Back-End sinnvoll:

- Front-End: maschinenunabhängiger Teil (z.B. Syntaxanalyse, lexikalische Analyse, Typüberprüfung).
- Back-End: maschinenspezifischer Teil (z. B. I/O, Codegenerierung)

3.3 Worte, Zeichenketten, formale Sprachen

Eine Zeichenkette (String) ist endliche Folge von Symbolen aus einem Alphabet Σ. z.B. $\Sigma = \{a, b, c\}$, Dann sind $ac, aaaabcc$ Strings über Σ. Die Menge aller Worte über Σ bezeichnet man als Σ^*

ε bezeichnet die leere Zeichenkette.

Operationen auf Strings:

- Konkatenation von x und y, geschrieben $x \cdot y$, $x \cdot \varepsilon = \varepsilon \cdot x = x$
- Vervielfachung: $x^0 = \varepsilon, x^i = x^{i-1} \cdot x$. z.B. $a^4 = aaaa$.

Eine *formale Sprache* über Σ ist eine Teilmenge von Strings über einem Alphabet Σ, d.h. eine Teilmenge von Σ^*.

Beispiel 1. $\Sigma = x, y$. Dann sind $\{\varepsilon\}$, $\{xy, x, y\}$ formale Sprachen über dem Alphabet Σ.

Beispiel 2.

Präfix: "ban" ist Präfix von "banane"

Suffix: "fix" ist Suffix von "suffix"

Teilstring: "r" und "affe" sind Teilstrings von "raffen"

Teilfolge: "ran" ist Teilfolge von "raffen", "aaff" ist keine Teilfolge von "raffen"

Beispiel 3.

- $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$, die Ziffernrepräsentation aller natürlichen Zahlen ist eine Sprache über Σ .
- $\Sigma = \{A,\ldots,Z,a,\ldots,z,0,1,2,3,4,5,6,7,8,9,),(,\ldots\}$. Dann ist der Quell-code als String aller möglichen Haskell-Programme eine formale Sprache über Σ.

Welche Möglichkeiten gibt es solche Sprachen zu beschreiben?

- Aufzählen der Wörter der Sprache
- reguläre Ausdrücke
- kontextfreie Grammatik
- kontextsensitive Grammatik
- Syntaxdiagramme
- Schreiben eines Programms, das alle Wörter erzeugt.
- Schreiben eines Programms, das alle Wörter erkennt.

3.3.1 Operationen auf formalen Sprachen:

- Konkatenation von L und S geschrieben LS, ist die Menge $\{ls \mid l \in L \text{ und } s \in S\}$.
- Vervielfachung: $L^0 = \{\varepsilon\}, L^i = L^{i-1}L$.
- Kleene-Abschluß L^*: Vereinigung aller Vervielfachungen von L, d.h. alle Worte, die durch Hintereinanderstellen von beliebig vielen Worten aus L hergestellt werden können.

$$L^* := \bigcup_{i=0}^{\infty} L^i$$

Positiver Abschluß:

$$L^+ := \bigcup_{i=1}^{\infty} L^i$$

3.4 Kontextfreie Grammatik

Eine *kontextfreie Grammatik (CFG)* ist ein 4-Tupel $G = (N, T, P, \sigma)$ mit

1.) N: endliche Menge von *Hilfszeichen (Nonterminals)*
2.) T endliche Menge von *Terminalzeichen (Terminals)*, das eigentliche *Alphabet*.

3.) $N \cap T = \emptyset$

4.) $\sigma \in N$ ist ein ausgezeichnetes Hilfszeichen *(Startzeichen, Axiom)*

5.) $P \subseteq N \times (N \cup T)^*$ eine endliche Menge (Produktionensystem)

$(A, w) \in P$ heißt Produktion oder Regel von G. Diese schreibt man auch $A \to w$ oder $A ::= w$.

Diese Grammatik nennt man *kontextfrei*, da die linke Seite von Regeln ein einziges Nichtterminal ist.

Wenn beliebige Strings in $(N \cup T)^*$ als linke Seite erlaubt wären, dann wäre diese nicht kontextfrei : Man nennt diese *unbeschränkte Grammatik, Semi-Thue-Systeme*. Man nennt die Grammatik *kontextsensitiv*, wenn in jeder Regel $a \to w$ gilt, daß $|a| \leq |w|$.

Konvention:

Nichtterminale: Erster Buchstabe groß, fett gedruckt statt $\langle$ Ziffer $\rangle$ schreiben wir **Ziffer**

Terminale: Operatoren: $+, - \ldots$, Ziffern: $1, 2, 3, 4, \ldots$, Buchstaben, Sonderzeichen, aber auch selbstdefinierte andere Zeichen.

Beispiel 4. Für kontextfreie Grammatik "Ausdruck":

Ausdruck	::=	**Ausdruck + Ziffer**
Ausdruck	::=	**Ausdruck - Ziffer**
Ausdruck	::=	**Ziffer**
Ziffer	::=	$0 \mid \ldots \mid 9$

Statt der drei Regeln können wir auch schreiben:

Ausdruck	::=	**Ausdruck + Ziffer**| **Ausdruck − Ziffer**|**Ziffer**
Ziffer	::=	$0 \mid \ldots \mid 9$

3.4.1 Herleitungen

Der Zusammenhang zwischen Grammatik und der durch sie festgelegten Sprache ergibt sich durch den Begriff der *Herleitung*.

Definition 1.

- w' kann aus w *direkt hergeleitet* werden, gdw $w = \alpha\gamma\beta$, $w' = \alpha\delta\beta$, und $\gamma ::= \delta$ eine Produktion ist.

  (Schreibweise $w \to w'$)

- $w_1 \to w_n$ gdw. es eine Folge $w_1 \ldots, w_n$ von Strings (aus Terminalen und Nichtterminalen) gibt, so daß jeweils $w_{i-1} \to w_i, 2 \leq i \leq n$.

Die zugehörige Sprache zu einer Grammatik $L(G)$ ist die Menge der Strings w aus Terminalsymbolen, die aus dem Startsymbol σ *hergeleitet* werden kann.

Schreibweise: $\sigma \to^* w$.

Beispiel 5.

$$\begin{array}{lll}\textbf{Ausdruck} & ::= & \textbf{Ausdruck} + \textbf{Ziffer} \mid \textbf{Ausdruck} - \textbf{Ziffer} \mid \textbf{Ziffer} \\ \textbf{Ziffer} & ::= & 0 \mid \ldots \mid 9\end{array}$$

Das Startsymbol der Grammatik ist **Ausdruck**, Terminalsymbole sind nur die Ziffern $0|\ldots|9$.

Eine Herleitung ist z.B.: **Ausdruck** $\rightarrow$ **Ausdruck** + **Ziffer** $\rightarrow$ **Ziffer** + **Ziffer** $\rightarrow$ 1+ **Ziffer** $\rightarrow 1+2$.

D.h. **Ausdruck** $\rightarrow^* 1+2$.

3.4.2 Rechts-und Linksherleitungen

Ersetzt man bei einer Herleitung immer nur das linkeste Nichtterminal, so ergibt sich eine **Linksherleitung**.

Ersetzt man immer nur das rechteste Nichtterminal, ist die Herleitung eine **Rechtsherleitung**.

Beispiel 6. Das Startsymbol der Grammatik ist **Ausdruck**, Terminalsymbole sind nur die Ziffern $0|\ldots|9$.

Eine mögliche Linksherleitung ergibt sich wie oben: **Ausdruck** $\rightarrow$ **Ausdruck** + **Ziffer** $\rightarrow$ **Ziffer** + **Ziffer** $\rightarrow$ 1+ **Ziffer** $\rightarrow 1+2$.

D.h. **Ausdruck** $\rightarrow^* 1+2$.

Eine mögliche Rechtsherleitung ist (des gleichen Strings): **Ausdruck** $\rightarrow$ **Ausdruck** + **Ziffer** $\rightarrow$ **Ausdruck** +2 $\rightarrow$ **Ziffer** +2 $\rightarrow 1+2$.

Man sieht, daß es auch Herleitungen gibt, die weder Rechts- noch Links-Herleitungen sind.

Beispiel 7.

$$\begin{array}{lll}\mathbf{A} & ::= & \mathbf{A} + \mathbf{A} \\ \mathbf{A} & ::= & \mathbf{A} - \mathbf{A} \\ \mathbf{A} & ::= & 0 \mid \ldots \mid 9\end{array}$$

Rechtsherleitungen von $1-2+3$:

$\mathbf{A} \rightarrow \mathbf{A} - \underline{\mathbf{A}} \rightarrow \mathbf{A} - \mathbf{A} + \underline{\mathbf{A}} \rightarrow \mathbf{A} - \underline{\mathbf{A}} +3 \rightarrow \underline{\mathbf{A}} -2+3 \rightarrow 1-2+3$.

Dies würde dem Ausdruck $(1-(2+3))$ entsprechen.

$\mathbf{A} \rightarrow \mathbf{A} + \underline{\mathbf{A}} \rightarrow \underline{\mathbf{A}} +3 \rightarrow \mathbf{A} - \underline{\mathbf{A}} +3 \rightarrow \underline{\mathbf{A}} -2+3 \rightarrow 1-2+3$.

Das entspricht $((1-2)+3)$.

Linksherleitungen von $1-2+3$:

$\mathbf{A} \rightarrow \underline{\mathbf{A}} + \mathbf{A} \rightarrow \underline{\mathbf{A}} - \mathbf{A} + \mathbf{A} \rightarrow 1- \underline{\mathbf{A}} + \mathbf{A} \rightarrow 1-2+ \underline{\mathbf{A}} \rightarrow 1-2+3$.

Das entspricht $(1-2)+3)$

$\mathbf{A} \rightarrow \underline{\mathbf{A}} - \mathbf{A} \rightarrow 1- \mathbf{A} \rightarrow 1- \underline{\mathbf{A}} + \mathbf{A} \rightarrow 1-2+ \underline{\mathbf{A}} \rightarrow 1-2+3$

Das entspricht $(1-(2+3))$.

In diesem Beispiel sieht man, was mit mehrdeutigen Ausdrücken passieren kann. Je nach Grammatik werden diese mehrdeutigen Ausdrücke als Strings der Sprache (mittels Herleitungen) erkannt, aber auf verschiedene Arten.

3.5 Herleitungs-Bäume, Parse-Bäume

Gegeben sei eine kontextfreie Grammatik G.

Ein *Herleitungsbaum, (Parse-baum, derivation tree, parse tree)* B eines Wortes w ist ein markierter Baum, so daß

1. Jeder Knoten ist mit einem Element aus $\{\varepsilon\} \cup N \cup T$ markiert.
2. Die Wurzel ist mit σ markiert,
3. innere Knoten sind mit Nichtterminalen markiert
4. wenn ein Knoten die Markierung A (ein Nichtterminal) hat und seine Töchter die Markierungen $X_1, \ldots, X_k$ (in dieser Reihenfolge), dann ist $A ::= X_1 \ldots X_k$ eine Regel in P.
5. $frontier(B) = w$. D.h. das an den Blättern von links nach rechts repräsentierte Wort ist genau w

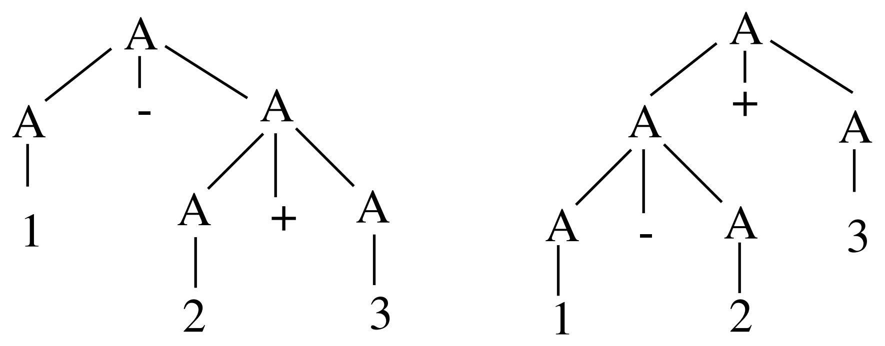
Es gilt:

- zu jeder Herleitung eines Wortes gibt es einen zugehörigen Herleitungsbaum.
- Ein Herleitungsbaum faßt mehrere Herleitungen eines Wortes der Sprache aus dem Startsymbol zusammen. Jeder mögliche Durchlauf der Knoten, der Väter vor den Söhnen durchläuft, entspricht einer Herleitung.
- Zu jedem Herleitungsbaum gibt es genau eine Rechtsherleitung und genau eine Linksherleitung. Linksherleitung entspricht Linksdurchlauf des Herleitungsbaumes. Rechtsherleitung entspricht Rechtsdurchlauf des Herleitungsbaumes.
- Meist wird die Bedeutung (oder die Übersetzung) eines Programms anhand des Herleitungsbaumes festgelegt.

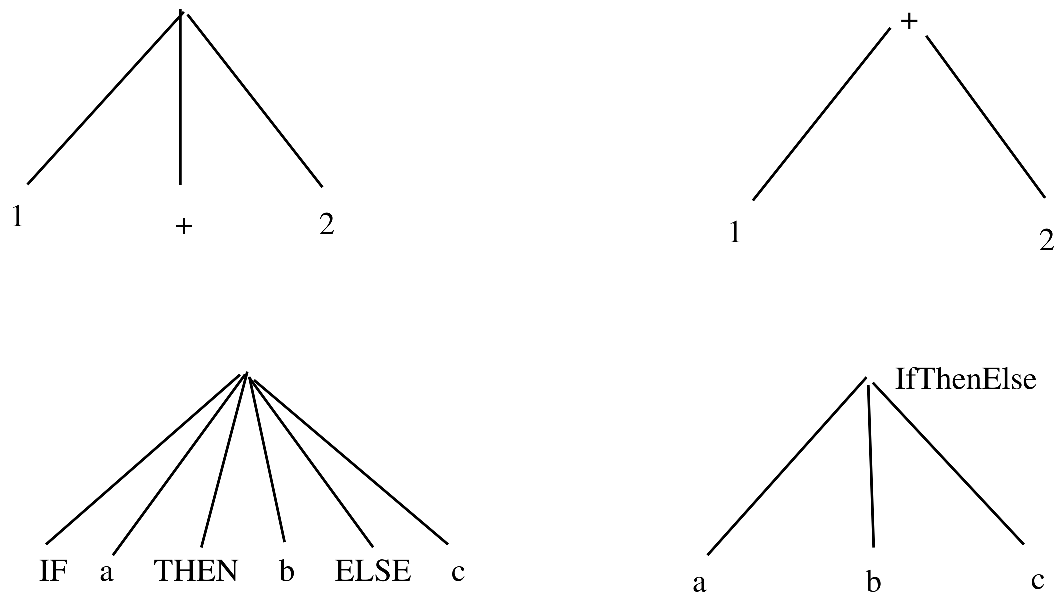
Beispiel 8. Beispielgrammatik, die Herleitungsbäume und die Mehrdeutigkeit demonstriert.

$$\begin{array}{lll} \mathbf{A} & ::= & \mathbf{A} + \mathbf{A} \\ \mathbf{A} & ::= & \mathbf{A} - \mathbf{A} \\ \mathbf{A} & ::= & 0 \mid \ldots \mid 9 \end{array}$$

Zwei verschiedene Herleitungsbäume für $1 - 2 + 3$ sind:



Statt Herleitungsbäumen verwendet man auch eine kompakte Darstellungsweise (des Baumes, die von der genauen Grammatik absieht, sogenannte Syntaxbäume. Im allgemeinen geht das Verfahren so vor: 1) Knoten werden mit dem Namen der Regel markiert, wobei auch Operatornamen wie $+, -, *$ verwendet werden. 2) Schlüsselworte, die keine Bedeutung tragen, und durch den namen der Regel bereits erfaßt sind, werden weggelassen. Folgende zwei Beispiele sollen dies illustrieren:



Eine Festlegung der Semantik der Worte einer formalen Sprache, die durch eine Grammatik gegeben ist, wird i.a. über den Herleitungsbaum definiert. D.h. man ordnet jedem Herleitungsbaum eine Bedeutung zu. Obiges Beispiel erlaubt zwei Herleitungsbäume für das Wort $1-2+3$, wobei die Bedeutung als Ausdruck einmal $(1-(2+3))$ und einmal $((1-2)+3)$ ist. Diese Bedeutungen sind offenbar verschieden. Nur der erste Baum entspricht der erwarteten Bedeutung. D.h. die Festlegung einer eindeutigen Semantik über den Herleitungsbaum ist für diese Grammatik nicht möglich.

Allerdings gibt es auch Situationen, in denen mehrere Herleitungsbäume pro Wort kein Problem darstellen, nämlich dann, wenn die Semantik trotzdem eindeutig ist. Zum Beispiel die zwei Herleitungsbäume zu $1 + 2 + 3$ bzgl. der gleichen Grammatik ergeben als Bedeutungen $((1+2)+3)$ und $(1+(2+3))$. Diese beiden Ausdrücke sind jedoch gleich, da $+$ assoziativ ist. Im allgemeinen muß man jedoch Mittel und Wege finden, um für bestimmte Ausdrücke Eindeutigkeit zu erzielen. Dazu kann man den Grammatikbegriff einschränken:

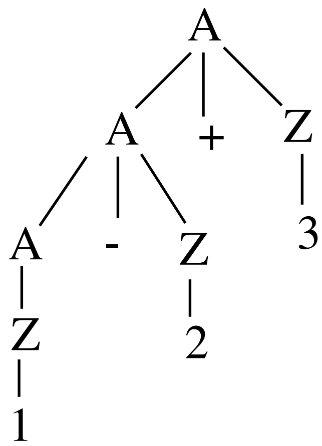
Definition 2. Eine Grammatik G heißt *eindeutig*, wenn es für jedes Wort der zugehörigen Sprache $L(G)$ genau einen Herleitungsbaum gibt.

In einer eindeutigen Grammatik gibt es genau eine Rechts- (Links-)herleitung für jedes Wort in $L(G)$.

Beispiel 9. Beispiel einer eindeutigen Grammatik mit der gleichen erzeugten Sprache wie Beispiel 8

$$\begin{array}{lll} \mathbf{A} & ::= & \mathbf{A} + \mathbf{Z} \\ \mathbf{A} & ::= & \mathbf{A} - \mathbf{Z} \\ \mathbf{Z} & ::= & 0 \mid \ldots \mid 9 \end{array}$$

Der einzige (d.h eindeutige) Herleitungsbaum für $1 + 2 - 3$ ist:



Diese Grammatik ist eindeutig.

Argumentation: Mit Induktion nach der Anzahl der Anwendungen der Regeln für $+, -$, sieht man, daß alle Worte der Sprache $L(G)$ folgende Form haben: $m_1 op_1 m_2 \ldots op_n m_n$ mit $m_i \in \{0, \ldots, 9\}, 1 \leq n$. Man sieht dann auch, daß es nur eine Rechtsherleitung für jedes solche Wort geben kann.

3.6 Syntaktische Analyse (Parsen)

Gegeben:	eine kontextfreie Grammatik G und ein String w.
Fragen:	gehört w zu $L(G)$?
	Welche Bedeutung hat w?
Vorgehen:	Konstruiere Herleitungsbaum zu w

Beispiel 10.

Gegeben	Syntax einer Programmiersprache und ein Quelltext.
Frage:	ist dies ein syntaktisch korrektes Programm?
	Was soll dieses Programm bewirken ?
Aufgabe:	Ermittle "Bedeutung" des Programms,
	d.h. konstruiere Herleitungsbaum, (bzw. Syntaxbaum)

Für jede kontextfreie Grammatik kann man einen Algorithmus zum Parsen angeben, der asymptotische Laufzeit $O(n^3)$ in der Anzahl n der Eingabesymbole hat. Für praktische jede Programmiersprache gibt es einen Parser, der effizient arbeitet, d.h. lineare oder fastlineare Laufzeit hat.

Wir wollen hier Parse-Methoden besprechen, die entweder einfach zu implementieren oder effizient sind. Diese gehen im allgemeinen so vor, daß die

Eingabe ein Strom (ein File, eine Liste) von Zeichen ist, der zeichenweise abgearbeitet wird, d.h. von links nach rechts, wobei oft noch das nächste Zeichen (einige der nächsten Zeichen) zur Steuerung mitverwendet werden.

Es gibt prinzipiell zwei Vorgehensweisen:

Top-Down : Das ist der Versuch, eine Herleitung vorwärts, vom Startsymbol aus, zu bilden ("forward-chaining")

Bottom-Up : Versuch, eine Herleitung rückwärts, vom Wort aus, zu bilden . ("backward-chaining").

Man unterscheidet bei der internen Vorgehensweise auch noch:

R : Konstruktion einer Rechtsherleitung

L : Konstruktion einer Linksherleitung

Beispiel 11.

$$\begin{array}{lll} \mathbf{S} & ::= & \mathbf{AB} \\ \mathbf{A} & ::= & 0 \mid 1 \\ \mathbf{B} & ::= & 8 \mid 9 \end{array}$$

Frage: Kann "09" aus dieser Grammatik hergeleitet werden?

Top-down: Hierzu muß man mit dem Startsymbol **S** anfangen und die Produktionen raten. Man sollte den String als Steuerung mitverwenden, um unsinnige Suche auszuschließen. Es ist unproblematisch, von links anzufangen, die Produktionen zu raten.

Ziel	09	09	9	ε
Herleitung	**S**	**AB**	0**B**	0w

Man beachte, daß bei bei dieser Vorgehensweise der String "09" von links nach rechts zeichenweise abgearbeitet wird und (bei dieser Grammatik) jedes Zeichen eindeutig die anzuwendende Produktion bestimmt. Im allgemeinen kann man mehrere Regeln anwenden, so daß die Suche verzweigt.

Bottom-up: Hierbei muß man die Regeln rückwärts auf den gegebenen String anwenden und versuchen, das Startsymbol der Grammatik zu erreichen. Auch hier kann man von links anfangen:

$09 \leftarrow \mathbf{A}9 \leftarrow \mathbf{AB} \leftarrow \mathbf{S}$

Problem auch hier: Man kann evtl. mehrere Regeln anwenden. Man muß außerdem den Teilstring raten, auf den eine Produktion (rückwärts) angewendet werden soll.

In beiden Fällen ergibt sich bei dieser einfachen Grammatik derselbe Herleitungsbaum.

Beispiel 12. Die Herleitung gelingt nicht immer sofort. Möglicherweise müssen mehrere Regeln ausprobiert werden.

$$\begin{array}{lll} \mathbf{S} & ::= & \mathbf{A} \mid \mathbf{B} \\ \mathbf{A} & ::= & 0\mathbf{A} \mid 1 \\ \mathbf{B} & ::= & 0\mathbf{B} \mid 2 \end{array}$$

Kann "002" aus dieser Grammatik hergeleitet werden?

Ziel	002	002	02	2	
Herleitung	**S**	**A**	0**A**	00**A**	?

"002" kann nur aus **B** hergeleitet werden:

Ziel	002	002	02	2	
Herleitung	**S**	**B**	0**B**	00**B**	002

Ein Parser müßte bereits beim ersten Zeichen von "002" wissen, welche Alternative (**A** oder **B**) weiterverfolgt werden soll.

Mißlingt eine Herleitung, so muß ein Parser zurücksetzen ("backtracking").

3.7 Rekursiv absteigende Parser

Diese Methode der Syntaxanalyse ist direkt an der Grammatik orientiert und versucht, in einem Top-Down Verfahren die Regeln anwendbar zu machen.

Eigenschaften:

1. Top-down Parser

2. für jedes Nichtterminal existiert ein Parser, der rekursiv andere Parser zu anderen Nichtterminalen aufrufen kann. Der Parser zum Startsymbol ist der Parser zur Grammatik.

3. Im Parser zu einem Nichtterminal N werden alle Produktionen zu N nacheinander ausprobiert. Bei Fehlschlag wird die nächste Produktion ausprobiert. Wenn keine anwendbar ist, gibt es keinen Parse: es wird "Fehlschlag" zurückgegeben.

4. Das zu analysierende Wort (Programm) wird von links nach rechts abgearbeitet, manchmal wird zurückgesetzt, d.h. es wird versucht, eine Linksherleitung zu konstruieren.

Probleme:

- i.a. ist der Zeitaufwand exponentiell.

- Die Methode terminiert nicht für bestimmte (linksrekursive) Grammatiken, obwohl eine Herleitung existiert:
  z.B. **A** ::= **A**+**A** | **A**-**A** | 1 | ... | 9

  Analysiere $1 + 1$: nur die erste Regel wird (zyklisch) versucht:
  (**A**,1+1) → (**A**+**A**,1+1) → (**A**+**A**+**A**, 1+1) → ...

Ein Vorteil der Methode ist, daß sie sich leicht implementieren läßt, so daß man für eine große Menge von Grammatiken einen Parser hat. Dies gilt auch für nicht kontextfreie Grammatiken.

Die Syntax von (Programmier-)Sprachen ist i.a. so, daß man Grammatiken angeben kann, so daß man ohne aufwendige Suche und Zurücksetzen die Programme analysieren kann.

Frage? Kann man eine Klasse von praktisch nützlichen Grammatiken angeben, und eine verbesserte Parsemethode, deren Sätze von einem Parser linear durch einmaliges Abarbeiten analysiert werden können?

3.7.1 funktionale Kombinator-Parser

Diese verwenden die Methode des rekursiven Abstiegs und haben deshalb die gleichen Probleme und Eigenschaften. Der Vorteil dieser Methode liegt in der relativ leichten Verständlichkeit und der 1-1-Übersetzung in den Programmcode für den Parser.

Die Idee ist folgende: Man schreibt für jedes Nichtterminal eine Funktion (Parser), die von der Eingabe etwas konsumiert, den Rest der Eingabe und das Resultat zurückliefert. Wenn R der Typ des Resultats ist, dann ist die Parsefunktion vom Typ.

Typ: `String -> (String,R)`

Da rekursiv absteigende Parser auch Zurücksetzen sollen, wird einfach eine Liste der möglichen Paare: Rest und Resultate als Antwort geliefert. Die verzögerte Auswertung sorgt dann für die richtige Reihenfolge der Auswertung.

Der Typ ist somit etwas allgemeiner:

Typ: `String -> [(String,R)]`

Es gibt eine Reihe von vordefinierten Kombinatoren (Funktionen auf Parsern), die es erlauben eine Grammatik direkt hinzuschreiben:

```
infixr 6 <*>, <*, *>
infixr 4 <|>, <!>
infixl 5 <@

type Parser a b = [a] -> [([a],b)]

--    erkennt ein Zeichen
symbol :: Eq s => s -> Parser s s
symbol a []                     = []
symbol a (x:xs)  | a ==x        = [(xs,x)]
                 | otherwise    = []

--    erkennt einen String
token :: Eq s => [s] -> Parser s [s]
--  token :: Eq s => [s] -> Parser s [s]
token k xs | k == (take n xs)   = [(drop n xs, k)]
           | otherwise          = []
                                  where n = length k

--   testet ein zeichen der Eingabe
satisfy :: (s -> Bool) -> Parser s s
satisfy p [] = []
satisfy p (x:xs) = [(xs,x) | p x]

epsilon :: Parser s ()
```

```
epsilon xs = [(xs,())]

--  immer erfolgreich
succeed :: r -> Parser s r
succeed v xs = [(xs,v)]

--  immer fehlschlagend
pfail :: Parser s r
pfail xs = []

 --   Sequenzkombinator
(<*>) :: Parser s a -> Parser s b -> Parser s (a,b)
(p1 <*> p2) xs = [(xs2, (v1,v2))
                 | (xs1,v1) <- p1 xs,
                   (xs2,v2) <- p2 xs1]

--   Alternativkombinator
(<|>) :: Parser s a -> Parser s a -> Parser s a
(p1 <|> p2) xs = p1 xs ++ p2 xs

--  Alternativkombinator2:  nimmt nur das erste Ergebnis einer Alternative:
(<!>) :: Parser s a -> Parser s a -> Parser s a
(p1 <!> p2) xs = take 1 (p1 xs ++ p2 xs)

--  ignoriert blanks am Anfang
sp :: Parser Char a -> Parser Char a
sp p = p . dropWhile (== ' ')

-- eliminiert alle "whitespaces"
wsp :: Parser Char a -> Parser Char a
wsp p = p . filter (\x -> not (x == ' ' || x == '\n'))

--  testet, ob die ganze Eingabe konsumiert wurde
just :: Parser s a -> Parser s a
just p = filter (null . fst) . p


--  Operation auf dem Ergebnis des Parse
(<@) :: Parser s a -> (a-> b) -> Parser s b

(p <@ f) xs = [(ys, f v) | (ys,v) <- p xs]

--  ignoriert rechtes Ergebnis
(<*) :: Parser s a -> Parser s b -> Parser s a
p <* q = p <*> q <@ fst

--  ignoriert linkes Ergebnis
```

```
(*>) :: Parser s a -> Parser s b -> Parser s b
p *> q = p <*> q <@ snd

list (x,xs) = x:xs

--  erkennt Folge. d.h. entspricht *
many :: Parser s a -> Parser s [a]
many p = p <*> many p <@ list
        <|> succeed []

many1 p = p <*> many p <@ list

digit :: Parser Char Int
digit = satisfy isDigit <@ f
       where f c = ord c - ord '0'

--  erkennt Zahl
natural :: Parser Char Int
natural = many1 digit <@ foldl f 0
             where f a b = a*10 + b

--  Nimmt nur die erste (maximale) Alternative des many
--  Ist nur erlaubt, wenn die Grammatik diese Alternativen nicht benoetigt

manyex :: Parser s a -> Parser s [a]
manyex p = p <*> many p <@ list
        <!> succeed []

many1ex p = p <*> manyex p <@ list
option p = p <@ (\x->[x])
           <!> epsilon <@ (\x-> [])

--  Nimmt nur die erste (maximale) Alternative bei Zahlen
naturalex :: Parser Char Int
naturalex = many1ex digit <@ foldl f 0
             where f a b = a*10 + b

--  Erkennt Klammerung und ignoriert den Wert der Klammern
pack:: Parser s a -> Parser s b -> Parser s c -> Parser s b
pack s1 p s2 = s1 *> p <* s2

--  Erkennt Infix-Folge wie z.B. (1+2+3+4+5): Liste der Argumente
opSeqInf psymb parg = (parg  <*> many (psymb *> parg))  <@ list

paarf f = \(x,y) -> f x y
```

Beispiel 13. arithmetische Ausdrücke mit folgender Grammatik

```
--   Ex ::= Plus
--   Plus ::= SigZ | SigZ Plusrest
--   PlusRest = + SigZ PlusRest | eps
--   SigZ ::= B | - B
--   B ::= Z | (Ex )

plusausdruecke = just pAriEx

pAriEx :: Parser Char Int

pAriEx    = (wsp pAriPlus)

pArisymbol = naturalex
pAriBasis = pArisymbol <|> (pack (symbol '(')  pAriEx  (symbol ')'))
pAriSig   = pAriBasis <|> (symbol '-')  *> pAriBasis <@ (negate)
pAriPlus  = (opSeqInf (symbol '+') pAriSig) <@ sum
```

3.8 Rekursiv-prädiktive Parser

Dies sind optimierte rekursiv absteigende Parser mit folgender Eigenschaft:

Die Entscheidung, welche Produktion anzuwenden ist, hängt jeweils nur vom nächsten Symbol der Resteingabe ("Lookahead-Symbol") ab. Somit ist kein Zurücksetzen notwendig, und die Eingabe kann von links nach rechts abgearbeitet werden.

Die Zuordnung Lookahead-Symbol zu Regel muß eindeutig sein, d.h. nimmt man die Menge der rechten Seiten zu einem Nichtterminal **T**. Dann darf es abhängig vom nächsten Symbol nur genau eine anwendbare Regel geben. Die ε-Produktion spielt eine Sonderrolle: diese wird nur dann angewendet, wenn es für das Lookahead-Symbol keine rechte Seite gibt.

Allerdings kann man nicht für jede Grammatik oder Sprache einen rekursiv-prädiktiven Parser konstruieren.

Zur Vorbereitung der Optimierung muß man sich die Grammatik näher anschauen:

Man kann zu jedem Nichtterminal die Menge der möglichen Nichtterminale angeben, die als erstes Zeichen in einem hergeleiteten String auftreten können. Da auch ε auftreten könnte, berechnet man eine solche Menge für jedes Nichtterminal, wobei diese Menge auch ε enthalten kann. Danach kann man zu jedem Wort aus Terminalen und Nichtterminalen ebenfalls diese Menge angeben. Diese Menge nennt man $FIRST(w)$ für ein Wort w.

Dies kann man zur Vermeidung von backtracking (Zurücksetzen) verwenden: insbesondere zur Konstruktion eines rekursiv prädiktiven Parser:

Gibt es für **A** zwei Produktionen $\mathbf{A} ::= w_1$ und $\mathbf{A} ::= w_2$ und ist $\mathrm{FIRST}(w_1)$ disjunkt zu $\mathrm{FIRST}(w_2)$, so kann man anhand des ersten Symbols des zu parsenden Wortes die richtige Produktion auswählen. Damit ist auch eine frühe Fehlererkennung möglich: wenn es zu einem Nichtterminal, das gerade angewendet werden soll weder eine anwendbare Regel noch eine ε-Alternative gibt, dann hat der Parser einen Fehler erkannt.

Beispiel 14. Eine vereinfachte Grammatik für Ausdrücke, in der die Zahlen nur Ziffern sind und die als Operatoren nur $+, -$ kennt, soll als Beispiel dienen:

$$\begin{array}{lll} \mathbf{Expr} & ::= & \mathbf{Term\ Rest} \\ \mathbf{Rest} & ::= & +\ \mathbf{Term\ Rest} \mid -\ \mathbf{Term\ Rest} \mid \varepsilon \\ \mathbf{Term} & ::= & 0 \mid \ldots \mid 9 \end{array}$$

Berechne die FIRST-Mengen:

- FIRST(**Term**) $= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- FIRST(**Expr**) = FIRST(**Term**) $= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- FIRST(**Rest**) $= \{+, -, \varepsilon\}$
- FIRST($+$ **Term Rest**) $= \{+\}$, FIRST($-$ **Term Rest**) $= \{-\}$

Nach Ermittlung dieser Mengen kann man eine sinnvolle Fehlererkennung durchführen.

Beispiel 15. Ein Beispielparser, der zur einfachen Grammatik von oben gehört.

```
--  E ::= T R
--  R ::= + T R | - T R | epsilon
--  T ::= 0 |... | 9


-- Funktionen liefern Paar zurueck: (Resteingabe, Pbaum)

data Pbaum  = Pblatt Int
                   | PExp Pbaum Pbaum
                   | PRest Char Pbaum Pbaum
                   | PLeer
              deriving (Show, Eq)

parseTerm [] = error "Ziffer erwartet"
parseTerm (symbol:rest) =
         if not  (symbol `elem` ['0'..'9'] )
         then  error "Ziffer erwartet"
         else (rest,Pblatt (fst (head (reads [symbol]))))

parseRest [] = ([],PLeer)
parseRest (operator: rest) =
   if not  (operator `elem` ['+','-'] )
         then  error "+ oder - erwartet"
```

```
   else
     let (rest1,resultat1) = parseTerm rest
         (rest2,resultat2) = parseRest rest1
       in (rest2, PRest operator resultat1 resultat2)

parseExp [] = error "Ziffer erwartet"
parseExp (eingabe@(symbol: _)) =
   if not  (symbol `elem` ['0'..'9'] )
         then  error "Ziffer erwartet"
   else
     let (rest1,resultat1) = parseTerm eingabe
         (rest2,resultat2) = parseRest rest1
       in  (rest2, PExp  resultat1 resultat2)

test1 = parseExp "1 + 2 - 3"         --   blanks sind noch da!!
test2 = parseExp "1+2-3"
```

3.9 Operationen auf Grammatiken

Es gibt Veränderungen der Regelmenge einer Grammatik, die, unter Erhaltung der zugehörigen formalen Sprache, das Erzeugen eines Parsers erleichtert. Unter anderem kann manchmal eine Grammatik so umkonstruiert werden, daß sie in einem rekursiv-prädiktiven Parsern benutzt werden kann:

3.9.1 Elimination der Epsilon-Produktionen

Diese Operation erlaubt es, alle Produktionen zu eliminieren, deren rechte Seite ein ε ist. Das Problem dieser Produktionen ist, daß sie (beim Bottom-Up)-Parsing stets anwendbar sind, und dort den Suchraum vergrößern. Natürlich müssen wir vorher den Fall ausschließen, (bzw. ignorieren), daß $\varepsilon \in L(G)$.

Die Elimination geschieht in zwei Schritten:

- Feststellen, welche Nichtterminale ein ε in ihrer erzeugten Sprache haben (die löschbar sind), und für welche Nichtterminale die erzeugte formale Sprache genau $\{\varepsilon\}$ ist (die leeren Nichtterminale). Dies macht man mit einer sogenannten Fixpunktiteration: Zunächst ermittelt man die Nichtterminale A , für die es eine Regel $A \to \varepsilon$ gibt. Danach sukzessive diejenigen Nichtterminale mit einer Regel, deren rechte Seite nur aus solchen besteht.

- Erzeuge folgende neue Regelmenge: Wenn $A \to X_1 \ldots X_n$ eine Regel ist, dann alle Regeln $A \to \alpha_1 \ldots \alpha_n$, die folgender Bedingung genügen: Wenn X_i nicht löschbar, dann $\alpha_i = X_i$; wenn X_i löschbar, dann kann α_i sowohl ε als auch X_i sein; wenn X_i leer ist, dann muß $\alpha_i = \varepsilon$ sein; die neue rechte Seite darf nicht ε sein.

Diese Elimination erzeugt eine neue Grammatik mit der gleichen erzeugten formalen Sprache (bis auf ε). Die Menge der Regeln kann exponentiell wachsen.

Diese Operation verändert die Menge der Regeln, damit die Herleitungsbäume und kann damit eine an den Herleitungsbäumen festgemachte Semantik verändern. Oft ist die Semantik allerdings so definiert, daß die ε-Resultate keine Rolle spielen. d.h. daß diese wie ein neutrales Element im Herleitungsbaum wirken. In diesem Falle kann die Semantik bei der änderung der Grammatik in eindeutiger Weise mitgeändert werden: d.h. die Worte der Sprache erhalten die gleiche Bedeutung (äquivalente Übersetzung).

Beispiel 16. Eine Grammatik, die positive ganze Zahlen beschreibt ist:

Zahl	::=	**ZifferZahl** | ε
Ziffer	::=	1 | ... | 9 | 0

Wenn wir diese ε-frei machen, erhalten wir:

Zahl	::=	**ZifferZahl** | **Ziffer**
Ziffer	::=	1 | ... | 9 | 0

Ein Parsebaum für das Wort "12" bzgl der ersten Grammatik ist, wobei wir die Regeln mit der Zeilen-Nr. und Alternativen-nr. indizieren : $R_{1,1}(R_{2,1}(1), R_{1,1}(R_{2,2}(2), R_{1,2}(\varepsilon)))$.

Eine Semantik φ_1 für diese Parsebäume ist:

- $\varphi_1(x) := \phi_1(x, 0)$
- $\phi_1(R_{1,1}(x, y), s) := \phi_1(y, 10 * s + \varphi_1(x))$
- $\phi_1(R_{1,2}(x), s) := s$
- $\varphi_1(R_{2,j}(x) := wert(x)$

Man sieht, daß ε den Wert 0 erhält.

Eine Semantik φ_2 für die zweite Grammatik ist:

- $\varphi_2(x) := \phi_2(x, 0)$
- $\phi_2(R_{1,1}(x, y), s) := \phi_2(y, 10 * s + \varphi_2(x))$
- $\phi_2(R_{1,2}(x), s) := 10 * s + \varphi_2(x)$
- $\varphi_2(R_{2,j}(x) := wert(x)$

Der Parsebaum für "12" sieht jetzt so aus:
$R_{1,1}(R_{2,1}(1), R_{1,2}(R_{2,2}(2)))$.

Die Ersetzung der ε Produktion läuft darauf hinaus, daß folgende Ersetzung richtig ist: $\varphi(R_{1,1}(R_{2,2}(2), R_{1,2}(\varepsilon)) = \varphi((R_{2,2}(2)))$ D.h., daß der $\varphi(R_{1,2}(\varepsilon))$ keine Rolle spielt, d.h. dem neutralen Element der Addition (der 0) entspricht.

3.9.2 Elimination von Links-Rekursionen

Damit ein rekursiv absteigender Parser nicht in Endlosschleifen gerät, ist es erforderlich, daß er ab und zu ein Symbol der Eingabe liest. Wenn eine Funktion, die einem Nichtterminal entspricht, sich selbst direkt wieder aufruft, ist dies nicht der Fall: eine Endlosschleife ist die Folge. Es gibt auch Fälle verschränkter Rekursion, die aus dem gleichen Grund zur Nichtterminierung führen können.

Beispiel 17. Direkte Links-Rekursion $\mathbf{A} ::= \mathbf{A}w$

$$\begin{array}{lll} \mathbf{Expr} & ::= & \mathbf{Expr} + \mathbf{Term} \mid \mathbf{Term} \\ \mathbf{Term} & ::= & 0|\ldots|9 \end{array}$$

Ein terminierender, rekursiv absteigender Parser kann aus dieser Grammatik nicht direkt konstruiert werden, da die Funktion zu **Expr** sich sofort selbst wieder aufruft, ohne ein Zeichen der Eingabe zu konsumieren.

Eine Methode zur Abhilfe ist das Einführen neuer Nichtterminale:
$\mathbf{A} ::= \mathbf{A}\alpha \mid \beta$
wird umgewandelt in:

$$\begin{array}{lll} \mathbf{A} & ::= & \beta\mathbf{B} \\ \mathbf{B} & ::= & \alpha\ \mathbf{B} \mid \varepsilon \end{array}$$

Hierbei ist **B** ein neues Nichtterminal

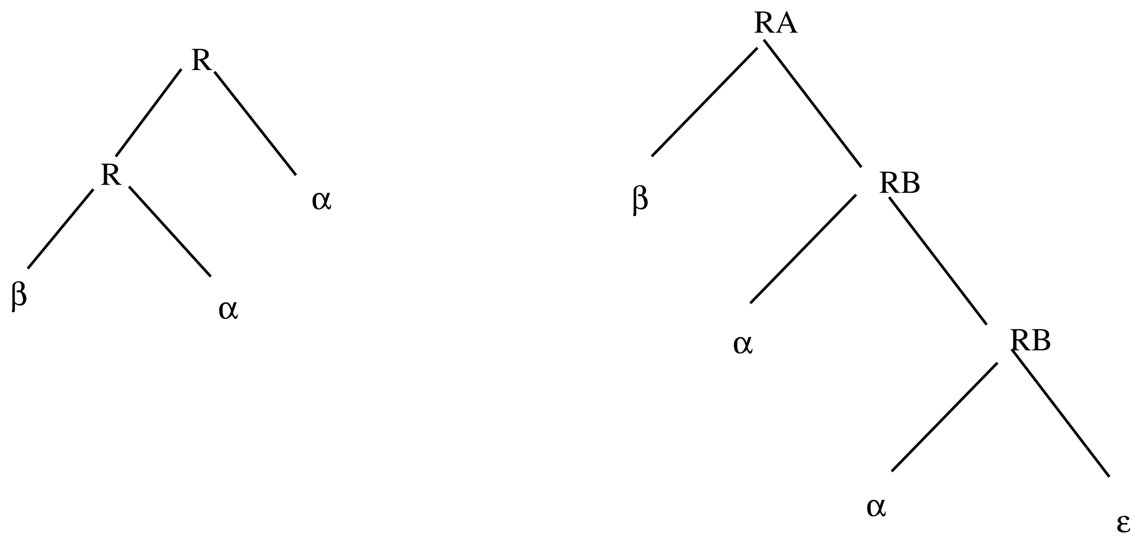
Wir geben eine Begründung für die Korrektheit dieser Umwandlung im Sinne der Erhaltung der formalen Sprache. Dazu nehmen wir an, daß β und α Worte aus Nichtterminalen sind.

Die zu **A** gehörige Sprache ist $\{\beta, \beta\alpha, \beta\alpha\alpha, \ldots\}$.

Die zu **B** gehörige Sprache ist $\{\varepsilon, \alpha, \alpha\alpha, \ldots\}$

Man sieht, daß sich die zu **A** gehörige formale Sprache nicht verändert hat.

Wir betrachten die Syntaxbäume vorher und nachher:



Eine automatische Behandlung der zugehörigen semantischen Abbildung zeigt, daß die Assoziativität eine Rolle spielen muß. Wenn man vorher weiß, daß der Operator, der zu R gehört, assoziativ ist, dann kann man die Operatoren zu R_A, R_B ebenfalls als R definieren mit dem Zusatz, daß ε auf ein neutrales Element abgebildet wird.

Definiert man als semantische Abbildung $\phi(R) = \phi(RA) = \phi(RB)$, so ergibt sich für den Herleitungsbaum vorher: $f(f(\beta', \alpha'), \alpha')$ und für den Herleitungsbaum nachher: $f(\beta', f(\alpha', f(\alpha', \varepsilon')))$. Wenn ε' ein neutrales Element für f ist und f assoziativ ist, dann ist der semantische Wert gleich.

Eine Umwandlung ohne semantische Betrachtung erfordert, daß man nachträglich die Semantik neu definieren muß, oder (im schlechten Fall) im Nachhinein die veränderte Semantik akzeptiert.

Beispiel 18. Zum Beispiel führt die automatische Elimination der Links-Rekursion in der folgenden Grammatik zu einer anderen Klammerung:

Expr ::= **Expr** − **Term** | **Term**
Term ::= 0| . . . |9

Vorher würde $1 - 2 - 3$ als $((1 - 2) - 3$ geklammert, danach als $(1 - (2 - 3))$. Das ergibt eine anderen Wert. Schaut man genauer hin, könnte man Abhilfe schaffen durch die Interpretation: $1 + (-2) + (-3)$, die einen assoziativen Operator verwendet.

Beispiel 19. Für eine indirekte Links-Rekursion:

A ::= **S**a | b
S ::= **B**c | **A**d | e
B ::= f . . .

Hier wird ein rekursiver Parser bei manchen (nicht allen) Worten, die zu $L(G)$ gehören, in eine Schleife geraten. Z.B. "fa" ist unproblematisch, aber bei "bd" gerät ein rekursiv absteigender Parser in eine Schleife.

Allgemein ist eine Grammatik **links-rekursiv**, wenn es für ein Nichtterminal **A** und einen String α eine Herleitung $\mathbf{A} \rightarrow * \mathbf{A}\alpha$ gibt.

Jede Linksrekursivität kann durch Transformation der Grammatik beseitigt werden, unter Beibehaltung der erzeugten formalen Sprache. Diese Transformationen bewirken i.a. eine Veränderung der Semantik.

3.10 Beseitigung von Mehrdeutigkeiten

Klassisches Beispiel "dangling else":

Stmt ::= `if` **Expr** `then` **Stmt** `else` **Stmt**
| `if` **Expr** `then` **Stmt**
| **other**

Diese Grammatik ist mehrdeutig, denn der String
`if` E_1 `then if` E_2 `then` S_1 `else` S_2
hat zwei Parsebäume, die wir mit Klammerung andeuten:

- `if` E_1 `then` (`if` E_2 `then` S_1 `else` S_2)
- `if` E_1 `then` (`if` E_2 `then` S_1) `else` S_2

Diese Mehrdeutigkeit kann man syntaktisch eliminieren, d.h. die Grammatik so umbauen, daß die erzeugte Sprache die gleiche bleibt, aber die Mehrdeutigkeit verschwindet. Das Problem, welche Semantik zugeordnet werden soll, muß ebenfalls gelöst werden: es gibt in diesem Fall die Vereinbarung, daß das `else` zum letzten freien `then` gehört.

In diesem Fall kann man die Elimination der Mehrdeutigkeit dadurch erreichen, daß man zwischen `then` und `else` kein halbes Statement erlaubt:

S	::=	**HIF** | **GIF**
HIF	::=	`if` **Expr** `then` **S**
GIF	::=	`if` **Expr** `then` **GIF** `else` **S** | **other**

3.11 Links-Faktorisierung

Ein weiteres Problem, das einen effizienten Einsatz rekursiv absteigender Parser behindert, sind die gleichen Anfänge verschiedener Regelalternativen: z.B.läßt sich manchmal mittels FIRST nicht entscheiden, welche Produktion anzuwenden ist. Die gleichen Anfänge bewirken ein zu weites Zurücksetzen. Das Zusammenfassen der Anfänge der Regelalternativen erlaubt einen schnelleren Übergang zu anderen Alternativen mit nur kurzem Zurücksetzen.

A ::= $\alpha\beta_1 \mid \alpha\beta_2$

wird umgewandelt in:

A ::= α**B**

B ::= $\beta_1 \mid \beta_2$

Beispiel 20.

H	::=	`if` **Expr** `then` **H**
		| `if` **Expr** `then` **G** `else` **H**

kann man umwandeln in

Ifethen	::=	`if` **Expr** `then`
H2	::=	**H** | **G** `else` **H**
H	::=	**Ifethen H2**

Dies optimiert rekursiv-absteigende Parser, da unnötiges Zurücksetzen verhindert wird. Eine Semantik aufbauend auf Herleitungsbäumen kann erhalten werden, da es sich hierbei nur um eine Zusammenfassung von Herleitungsmöglichkeiten handelt.

Man kann auch gleiche Anfänge rechter Seiten für verschiedene Nichtterminale zusammenfassen:

S	::=	**HIF** | **GIF**
HIF	::=	`if` **Expr** `then` **S**
GIF	::=	`if` **Expr** `then` **GIF** `else` **S**

wird zu:

S	::=	**IET**
IET	::=	`if` **Expr** `then` **IET2**
IET2	::=	**HIF2** | **GIF2**
HIF2	::=	**S**
GIF2	::=	**GIF** `else` **S**
GIF	::=	`if` **Expr** `then` **GIF** `else` **S**

3.11.1 Bemerkungen zur Fehlererkennung

Da ein eingegebenes Programm oder eine LaTex-Datei normalerweise bei der ersten Eingabe nicht fehlerfrei sind, d.h. sie werden nicht als Wort der entsprechenden formalen Sprache erkannt, müssen Übersetzer einen Fehlerausgang haben und diesen Fehler melden. Diese Fehlererkennung, sollte spezifisch sein und die Rückmeldung so genau, daß es dem Programmierer, (Textschreiber) möglich ist, das Problem genau genug zu lokalisieren und zu beheben.

Eine gute Fehlererkennung bei rekursiv absteigenden Parsern mit Backtracking ist generell schwierig, denn ein Fehler beim Parsen wird erst erkannt, wenn es keine weitere Alternative zu einer Herleitung vom Startsymbol aus mehr gibt. D.h. ohne zusätzliche Information wie: "nach einem + kann nicht noch ein + kommen", die man auch dann verwenden darf, wenn es im Prinzip noch andere Möglichkeiten des Zurücksetzens gibt, haben diese Parser schlechte Fehlererkennungsmöglichkeiten. Z.B. ist der Fehlerort meist unklar.

Für rekursiv-prädiktive Parser, die kein Backtracking verwenden, ist die Fehlererkennung relativ einfach: wenn es für ein Lookahead-symbol keine Fortsetzung mehr gibt, wird abgebrochen und ein Fehler gemeldet. Die Stelle, an der der Fehler aufgetreten ist, läßt sich im Quelltext identifizieren. Die Fehlermeldung kann oft anhand der letzten Regel oder anhand des Zustandes des Programms spezifisch genug formuliert werden.

3.12 LR-Parser, Shift-Reduce-Verfahren

Eine andere allgemeine Methode, einen Syntaxanalyse zu implementieren, sind sogenannte LR-Parser (L: von links nach rechts; R: Rechtsherleitung). Diese Methode ist eine Bottom-Up Methode. Ein Zurücksetzen ist nicht erlaubt.

Betrachten wir zunächst ein Beispiel:

Beispiel 21. Gegeben sei die CFG:

$$
\begin{array}{lll}
\mathbf{E} & ::= & \mathbf{E} + \mathbf{E} \\
\mathbf{E} & ::= & \mathbf{E} * \mathbf{E} \\
\mathbf{E} & ::= & (\mathbf{E}) \\
\mathbf{E} & ::= & \mathrm{a} \mid \mathrm{b}
\end{array}
$$

Betrachte folgende Rechtsherleitung von `c+ b * a`:

$$
\begin{array}{lll}
\mathbf{E} & \to & \underline{\mathbf{E} + \mathbf{E}} \\
 & \to & \mathbf{E} + \underline{\mathbf{E} * \mathbf{E}} \\
 & \to & \mathbf{E} + \mathbf{E} * \underline{\mathrm{a}} \\
 & \to & \mathbf{E} + \underline{\mathrm{b}} * \mathrm{a} \\
 & \to & \underline{\mathrm{c}} + \mathrm{b} * \mathrm{a}
\end{array}
$$

Ein (*Handle*) (bzgl. einer Grammatik) ist ein Substring in einem Wort über $(T \cup N)^*$, das eine rechte Seite einer Regel ist, so daß es eine Rechtsherleitung gibt, die als letzte Ersetzung gerade dieses Handle mit dieser Regel eingefügt hat. Als Umkehrung wird ein solches Handle dann zur Konstruktion einer Rechtsherleitung in umgekehrter Reihenfolge verwendet.

Beachte, daß ein Handle nicht immer eindeutig ist, da es mehrere Rechtsherleitungen geben kann.

Beispiel Fortsetzung von Beispiel 21 . *Betrachte jetzt den String* `c+ b * a` *und versuche, eine Rechtsherleitung durch Ersetzen von Handles zu konstruieren.*

hergeleitetes Wort	*Handle*	*Produktion*
$c + b * a$	c	$\mathbf{E} \to c$
$E + b * a$	b	$\mathbf{E} \to b$
$E + E * a$	a	$\mathbf{E} \to a$
$E + E * E$	$\mathbf{E * E}$	$\mathbf{E} \to \mathbf{E * E}$
$E + E$	$\mathbf{E + E}$	$\mathbf{E} \to \mathbf{E + E}$
E		

3.12.1 Schiebe-Reduziere (Shift-Reduce)-Implementierung

Mit diesen Vorbereitungen kann man die sogenannte Shift-Reduce Analysemethode implementieren. Dazu benötigt man als Argumente die Eingabe (den Rest der Eingabe) und einen Stack, der den Anfang des bisher abgeleiteten Wortes enthält, so daß der Stack zusammen mit dem Rest das bisher hergeleitete Gesamtwort ergibt.

Die wesentlichen Aktionen der Analyse sind

Schieben: Lesen eines Zeichens der Eingabe und Ablage auf den Stack, und

Reduzieren: Ersetzen eines obersten Teil des Stacks, das Handle, mittels einer Produktion, rückwärts angewendet.

Dies soll am Beispiel verdeutlicht werden. Hierbei soll ein $ das Ende der Eingabe und den Boden des Stacks anzeigen.

Stack	Eingabe	Aktion
$	c + b*a$	schiebe
$c	+ b*a$	reduziere mit E ::= c
$E	+ b*a$	schiebe
$E+b	*a$	reduziere mit E ::= b
$E+E	*a$	schiebe
$E+E*	a$	schiebe
$E+E*a	$	reduziere mit E ::= a
$E+E*E	$	reduziere mit E ::= E*E
$E+E	$	reduziere mit E ::= E+E
$E	$	akzeptiere

D.h. der Parser kann 4 Aktionen durchführen:

Schieben: Zeichen von Eingabe auf Stack

Reduzieren: Handle, d.h. ein oberes Stück des Stacks durch ein Nichtterminal ersetzen.

Akzeptieren: Wenn auf dem Stack das Startsymbol steht und die Eingabe leer ist.

Fehlererkennung: Wenn weder Schiebe noch Reduzieraktion möglich.

Ein Shift-Reduce-Parser muß eindeutig wissen, was in welcher Situation zu tun ist. Als Quellen der Information kann er den Inhalt des Stacks und das erste Symbol (die ersten k Symbole) der Eingabe verwenden.

Bei Erzeugung einer Rechtsherleitung auf diese Weise muß man sich klarmachen, daß der Handle immer oben auf dem Stack erscheinen muß. Es ist niemals notwendig, Zeichen in die Eingabe zurückzuschieben:

Eine Rechtsherleitung ersetzt immer das rechteste Nichtterminal. Dies entspricht dem Moment des Reduzierens: ein Nichtterminal steht auf dem Stack. Eine Rechtsherleitung kann ein Nichtterminal durch ein Wort ersetzen, das ganz rechts ein Nichtterminal hat: das entspricht zwei Reduzieraktionen hintereinander. Wenn das Nichtterminal durch ein Wort mit rechten Terminalen ersetzt wird, dann entspricht das sovielen Schiebeaktionen wie rechts Terminale stehen. Eine ε-Produktion entspricht einer Ersetzung eines leeren Handles durch ein Nichtterminal.

Die allgemeine Umsetzung von Grammatiken in einen Shift-Reduce-Parser ist von Hand nicht einfach durchzuführen und ist nicht Gegenstand dieses Skripts. Für einige spezielle Beispielgrammatiken werden wir einen solchen Parser angeben.

Die Konflikte, die hierbei auftreten sind: 1) Schiebe-Reduziere-Konflikt: Es kann eine Situation entstehen, in der unklar ist, ob geschoben oder reduziert werden soll. 2) Reduziere-Reduziere-Konflikt: Es kann eine Situation entstehen, in der unklar ist, mit welcher Regel reduziert werden soll.

Diese Fehlermeldungen können auch während der Erzeugung eines Parser aus einer Grammatik durch einen LR-Parser-Generator auftreten, wobei hier gemeint ist, daß ein solcher Konflikt während des Parsens nicht ausgeschlossen werden kann, und somit (aufgrund der aktuellen Eingabe) ein passender Shift-Reduce Parser nicht konstruiert werden kann.

3.12.2 Operatorgrammatiken, Operator-Prioritäts-Syntaxanalyse

Eine *Operatorgrammatik* hat als kennzeichnende Merkmale, daß es keine ε-Produktion gibt und auch keine rechte Seite einer Produktion mit direkt benachbarten Nichtterminalen.

Beispiel 22. Die Grammatik E ::= $E + E \mid -E \mid E - E \mid E * E \mid E/E \mid (E) \mid E\hat{\ }E \mid E! \mid$ **Id** wobei **Id** ein Nichtterminal ist, das für Bezeichner steht, ist eine Operatorgrammatik.

Beachte, daß diese Grammatik mehrdeutig ist.

In Operatorgrammatiken verwendet man verschiedene Methoden, um Ausdrücke eindeutig zu machen: Man gibt den Operatoren weitere Eigenschaften:

- Stelligkeit: wieviele Argumente bindet der Operator? Im allgemeinen sind die Werte hier 1 oder 2.
- Infix, Prefix oder Postfix: welcher Ausdruck (welche Ausdrücke) bindet der Operator .

- Prioritäten von Operatoren : welcher Operator bindet stärker. Man verwendet i.a. natürliche Zahlen, um die Stärke der Priorität anzugeben.
- Assoziativität: Rechtsassoziativ oder linksassoziativ. Werden Ausdrücke bevorzugt rechts zuerst oder links zuerst geklammert? . Z.B. $a + b + c$ kann als $a + (b + c)$ (rechtsassoziativ) oder als $(a + b) + c$ (linksassoziativ) geklammert werden.

Obige Grammatik wird eindeutiger, wenn wir folgende, übliche Vereinbarungen treffen: $+, *, /,$ ^ sind zweistellige Infixoperatoren. $-$ kann zweistelliger Infix- oder einstelliger Prefixoperator sein. Die Priorität ist: ! vor ^ vor einstelligem $-$ vor $*, /$ vor $+, -$. Die zweistelligen Operatoren $+, *, /, -$ sind linksassoziativ, ^ ist rechtsassoziativ. Wir gehen davon aus, daß die Operatoren selbst keine Bezeichner sind. Die Klammersymbole werden getrennt behandelt.

Beispiel 23. $1 + 2!$ ^ 3 entspricht $1 + ((2!)$ ^ $3)$ $1 - 3 - 5 * 6!$ entspricht ((1-3) - (5*(6 !))

Man kann diese Vereinbarungen in eine CFG kodieren. D.h., die Sprache kann durch eine CFG erzeugt werden. Allerdings ist die Datengrundlage eines Shift-Reduce Parser nicht die volle Grammatik, sondern die Eingabe der Operatoren und deren Eigenschaften.

Beispiel 24. Als Beispiel wollen wir die Grammatik konstruieren, die zu einem Postfix ! mit hoher Priorität, dem einstelligen $-$ mit geringerer Priorität und dem Zeichen $+$ mit geringster Priorität gehört . **Zahl** sei das Nichtterminal für Zahlenkonstanten.

E	::=	**PlusE**
FakE	::=	**Zahl** | **FakE !** | (**PlusE**)
MinE	::=	**FakE** | **- MinE**
PlusE	::=	**MinE** | **PlusE + FakE**

Diese Grammatik kann z.B. $- - 1!!$ als $-(-((1!)!))$ erkennen. +-Sequenzen werden linksassoziativ geklammert. Vermutlich ist diese Grammatik eindeutig. Diese Grammatik ist linksrekursiv, kann aber mit den üblichen Mitteln frei von Linksrekursionen gemacht werden.

3.12.3 Vorgehen bei der Shift-Reduce Syntaxanalyse

Eine nicht vollständige Liste von möglichen Aktionen ist:

1. Die Eingabe wird von links nach rechts abgearbeitet. Auf dem Stack merkt man sich in jeweils einem Eintrag in der Reihenfolge der Eingabe:
    - die bisherigen Operatoren,
    - die offenen Klammern und
    - Bezeichner / erkannte Ausdrücke (d.h. Herleitungsbäume)

2. Wenn ein Bezeichner in der Eingabe ist, und ein Operator (auch Klammer) auf dem Stack: Schieben. In gewissen Fällen kann man danach reduzieren, Z.B., wenn $-$;E auf dem Stack steht.

3. Bei "(" , dann Schieben

4. Bei ")": Wenn der Stack noch offene Operatoren enthält, dann Reduzieren.

   Am Ende muß auf dem Stack "(" ; E stehen: Der Klammerausdruck (E) wird erkannt.

5. Wenn ein Operator op in der Eingabe: Suche letzten zweistelligen Operator.

   - Wenn dieser grössere Priorität hat, dann: Reduzieren auf dem Stack.
   - Wenn dieser gleiche Priorität hat und linksassoziativ, dann ebenfalls auf dem Stack reduzieren.
   - wenn niedrigere Priorität, dann Reduzieren.

Beispiel 25. Das Ende der Eingabe wird mit $ bezeichnet. Die Bezeichner sollen nur Ziffern sein.

$	1 - 3 - 5 * 6 ! $	S
$1	- 3 - 5 * 6 ! $	R
$E	- 3 - 5 * 6 ! $	S
$E -	3 - 5 * 6 ! $	S
$E - 3	- 5 * 6 ! $	R
$E - E	- 5 * 6 ! $	R
$E	- 5 * 6 ! $	S
$E -	5 * 6 ! $	S
$E - 5	* 6 ! $	R
$E - E	* 6 ! $	S
$E - E*	6 ! $	S
$E - E*6	! $	S
$E - E*6!	$	R
$E - E*E	$	R
$E - E	$	R
$E	$	

Erkannt wurde der Ausdruck: $((1 - 3) - (5 * (6!)))$

Bemerkung. Bemerkungen zu dieser Methode:

- Der Parser baut nicht direkt auf einer Grammatik auf. Deshalb kann es in der Praxis leicht passieren, daß eine angegebene Grammatik und die erkannte Sprache nicht zusammenpassen.

- Fehler in der Eingabe können gut erkannt werden: sobald es keine erlaubte Aktion mehr gibt, d.h. Schieben oder Reduzieren, kann ein Fehler gemeldet und abgebrochen werden.

- Mehrdeutigkeiten werden i.a. durch die Implementierung des Parsers aufgelöst. Bei gleicher Priorität "gewinnt" der frühere Operator.

- Zweideutigkeit von – (einstellig Prefix und zweistellig Infix) wird entschieden durch folgende Vereinbarung: wenn links von – kein Operand, dann einstellig, sonst zweistellig.
- Operatoren, die links mehr als ein Argument konsumieren, sind i.a. nicht zugelassen.
- Vorteil: Die Operatoren und deren Eigenschaften können vom Benutzer definiert werden, so wie dies z.B. in Haskell gemacht wird.
- Vorteil: Eingabe ist benutzerfreundlich
- Nachteil einer zu klammerfreien Eingabe: bei komplexer Eingabe und benutzerdefinierten Operatoren kann man die Bedeutung der Ausdrücke als Mensch nicht mehr so gut erkennen: Dies gilt auch noch, wenn ein starkes Typsystem noch eine semantische Prüfung macht, die unsinnige Ausdrücke vermeidet.

  Abhilfe: wenn man unsicher ist, mehr Klammern setzen.
- Es gibt Erweiterungen, die statt Prioritäten Prioritätsrelationen verwenden,: ¡ muß keine lineare oder partielle Ordnung mehr sein.
- in Haskell gibt es noch als Erweiterung die Möglichkeit, "(+)" als Ausdruck Argument zu verwenden.

3.12.4 Parsergeneratoren

Es gibt verschiedene Parsergeneratoren: Z.B. yacc. Diesem nachempfunden wurde der Parsergenerator "Happy" für Haskell und der Generator "Lucky". Diese haben als Eingabe im wesentlichen die kontextfreie Grammatik. Erzeugt wird ein (Shift-Reduce-) Parser für die eingegebene Grammatik.

3.13 Lexikalische Analyse , Tokenizer, Scanner

Dies ist der Teil eines Übersetzers, der den Zeichenstrom vorverarbeitet und die einfachen Bestandteile wie Zahlen, Bezeichner, Namen, Schlüsselworte, Strings schon erkennt, und in eine für den Übersetzer einfachere Form bringt. Die Ausgabe des Tokenizers ist eine Folge von Token (Strom von Token). Die Symboltabelle wird ebenfalls vom Tokenizer schon aufgebaut. Diese Unterteilung ist nicht zwingend, sondern eine pragmatische vorteilhafte Aufteilung.

Die Aufgaben eines Parsers werden hierdurch aufgeteilt in:

Scanner Zeichen (Symbol-) analyse

Parser Syntax-analyse:

I.a. entspricht ein Scanner einem Erkenner für reguläre Ausdrücke.

Ein Scanner liest ein Programm als lineare Zeichenkette ein und beseitigt z.B. Kommentare sowie Leer- und Tabulatorzeichen. Bestimmte Ketten von Zeichen werden zu Symbolen mit Attributen *token* zusammengefaßt. Ein solches Token kann ein komplexere Datenstruktur sein, bestehend aus:

- Markierung: Nur diese wird von der Grammatik des anschließenden Parsers erfaßt.
- Attribute: Diese werden in den Syntaxbaum mit aufgenommen.

Zum Beispiel kann ein Objekt bestehend aus der Markierung "Zahl" und als Attribut die Zahl 321 ein solches Token sein.

Dieser Strom von tokens ist dann die Eingabe des Parsers.

Eingabe	Tokenmarkierung	Attribut
123	Num	123
x1	Id	'x1'
$*$	Mult	Code($*$)
<=	Relop	Code(<=)

Die Trennung in Zeichenerkennung und syntaktische Analyse entspricht der folgenden Zerlegung einer Beispiel-CFG:

$$\text{Parser}\quad\left\{\begin{array}{lll} \mathtt{E} & ::= & \mathtt{E} + \mathtt{T} \mid \mathtt{T} \\ \mathtt{T} & ::= & \mathtt{T} + \mathtt{F} \mid \mathtt{F} \\ \mathtt{F} & ::= & (\mathtt{E}) \mid \mathtt{Zahl} \mid \mathtt{Bezeichner} \end{array}\right.$$

$$\text{Tokenizer}\quad\left\{\begin{array}{lll} \mathtt{Zahl} & ::= & \mathtt{Ziffer} \mid \mathtt{ZifferZahl} \\ \mathtt{Ziffer} & ::= & 0|1|2|3|4|5|6|7|8|9 \\ \mathtt{Bezeichner} & ::= & \mathtt{Char} \text{ — } \mathtt{Char\ Alphanum} \\ \mathtt{Alphanum} & ::= & \mathtt{Ziffer} \mid \mathtt{Char} \end{array}\right.$$

Der Parser kümmert sich dann im wesentlichen nur um Tokenmarkierung. d.h. das Alphabet des Parsers ist die Menge der Tokenmarkierungen. Diese Abstraktion, insbesondere die Auslagerung der Attribute hat als weiteren Vorteil, daß die vom Parser zu verarbeitende formale Sprache weniger Abhängigkeiten hat. Z.B. abstrahiert der Parser dann von den exakten Namen der Bezeichner und sieht dann auch nicht, ob die Deklaration von Variablen vor der Benutzung erfolgt, so daß die formale Sprache auf dieser Ebene kontextfrei ist.

Z.B. tatsächlicher Parseralphabet wäre {+,*,Z, B,),(}

Typ eines Scanners:	String -> [token]
Typ eines det. Teil-Parsers:	[token] -> ([token], Syntaxbaum)
Typ eines Teil-Parsers, der Zurücksetzen erlaubt:	[token] -> [([token], Syntaxbaum)]
Typ des Gesamt-Parsers:	[token] -> Syntaxbaum

3.13.1 Fehlererkennung des Scanners

Der Scanner erkennt nur eine geringe Auswahl der Fehler. Z.B. kann er folgende Fehler erkennen:

- bei der Erkennung von Zahlen syntaktisch falsch eingegebene erkennen. Z.B. könnte 123.a ein Fehler sein.
- beim Scannen von Strings falsche Escape-folgen, ein fehlendes Stringendezeichen, oder evtl. Längenüberschreitungen.

- Bezeichner, die nicht mit Buchstaben beginnen.
- Ungültige Symbole in bestimmten Kontexten: In Strings, in Bezeichnern ....

I.a. nicht erkennbar für einen Scanner: Klammerfehler, auch bei geschachtelten Kommentaren, falsche Schlüsselworte,

Bei der Implementierung eines Scanners wird i.a. der Eingabe-String nicht durch das physikalische Ende (EOF) beendet, sondern man testet ein extra Ende-Zeichen. Dasselbe gilt für die Weitergabe an den Parser: es wird am Ende des Tokenstroms ein eigenes Ende-Token angehängt. Dies vermeidet den Umstand, jeweils das Ende des Stroms abfragen zu müssen.

3.13.2 Scannen von Kommentaren und Strings

Ein String sei eine Zeichenkette mit linkem und rechtem Begrenzungssymbol ".

Ein bekanntes Problem ist: wie teilt man dem Scanner mit, daß bestimmte Zeichen in einem String als Zeichen stehen sollen: Z.B. daß "ein Zeichen innerhalb des Strings ist oder daß das Zeilenendezeichen im String vorkommen soll. Diese Möglichkeit ist sinnvoll und notwendig, wenn man z.B ein Programm erzeugen soll, daß selbst wieder Strings enthält.

Es gibt folgende sinnvollen Lösungsalternativen:

1. es gibt ein Escape-symbol \, mit dem man verschiedene Extrazeichen in einem String unterbringen kann. \"innerhalb eines Strings wird übersetzt in ". \\ innerhalb eines Strings wird übersetzt in \.
2. Verdopplung: ""innerhalb eines Strings wird übersetzt in ".

Beachte auch, daß das Entfernen von Leerzeichen (whitespace: blanks, Zeilenende, carriage return, Tabulatorzeichen vom Scanner erledigt wird. Allerdings darf der Scanner blanks in Strings nicht entfernen.

Scannen von Kommentaren:

Zeilenkommentare werden markiert durch ein Kommentar-anfangssymbol innerhalb einer Zeile, z.B. ; oder % oder −−.

geklammerte Kommentare Kommentare sind von Kommentar-begrenzern eingeschlossen. Z.B. Anfang: * , Ende: *\. Diese Kommentare kommen möglicherweise auch geschachtelt vor.

Ein Scanner, der zeilenweise liest, kann Zeilenkommentare eliminieren. Allerdings ist es normalerweise nicht die Aufgabe des Scanners, auch die geklammerten Kommentare zu entfernen.

Probleme:

- Zeilenkommentarzeichen innerhalb eines Strings zählen als Stringzeichen.
- geschachtelte Kommentare: Die zugehörige Sprache ist nicht regulär, d.h. der Scanner ist eigentlich überfordert, diese zu erkennen und zu eliminieren

Bemerkung. Es gibt auch Generatoren für Scanner (Z.B. lex). Allerdings ist es oft einfacher, einen Scanner selbst zu schreiben.

3.14 Reguläre Ausdrücke, endliche Automaten

3.14.1 Reguläre Grammatik

Die meisten Token von Programmiersprachen lassen sich durch reguläre Grammatiken beschreiben:

Definition 3. eine reguläre Grammatik ist eine kontextfreie Grammatik, bei der die Produktionen von der Form $\mathbf{A} ::= \mathrm{w}\mathbf{B}$, $\mathbf{A} ::= \mathrm{w}$ oder $\mathbf{A} ::= \mathbf{B}$ sind. Hierbei sind $\mathbf{A}$, $\mathbf{B}$ Nichtterminale und w ein String von Terminalen.

Es gilt: jede reguläre Sprache läßt sich durch eine reguläre Grammatik erzeugen. Und umgekehrt: die von regulären Grammatiken erzeugten Sprachen sind regulär.

Beispiel 26. Bezeichner, die mit einem Buchstaben beginnen, gefolgt von Buchstaben und Ziffern.

$$\begin{array}{lll} \mathbf{I} & ::= & \mathrm{a}\mathbf{R} \mid \ldots \mid \mathrm{z}\mathbf{R} \mid \mathrm{a} \mid \ldots \mid \mathrm{z} \\ \mathbf{R} & ::= & \mathrm{a}\mathbf{R} \mid \ldots \mid \mathrm{z}\mathbf{R} \mid 0\mathbf{R} \mid \ldots \mid 9\mathbf{R} \mid \mathrm{a} \mid \ldots \mid \mathrm{z} \mid 0 \mid \ldots \mid 9 \end{array}$$

Reguläre Grammatiken erzeugen reguläre Sprachen, die sich durch reguläre Ausdrücke darstellen lassen.

Beispiel 27.

$$\begin{array}{lll} \mathbf{B} & ::= & \mathrm{a} + \ldots + \mathrm{z} \\ \mathbf{Z} & ::= & 0 + \ldots + \mid 9 \\ \mathbf{I} & ::= & \mathbf{B}(\mathbf{B} + \mathbf{Z})^* \end{array}$$

3.14.2 Reguläre Ausdrücke

Sei Σ ein Alphabet. Die Menge der regulären Ausdrücke über Σ kann durch folgende Grammatik definiert werden:

$$\begin{array}{llll} \mathbf{R} & ::= & a & \text{für } a \in \Sigma \\ & & (\mathbf{R}) & \\ & & \varepsilon & \\ & & \mathbf{RR} & \\ & & \mathbf{R} + \mathbf{R} & \\ & & \mathbf{R}^* & \end{array}$$

Bei dieser Syntax wollen wir der Einfachheit erlauben, daß beim Nebeneinanderschreiben Klammern weggelassen werden können, wenn dies die Eindeutigkeit nicht beeinträchtigt. (Genaugenommen ist das dann eine andere Grammatik.)

Die Semantik ψ eines regulären Ausdrucks über Σ wird durch Zuordnung von Mengen von Strings über Σ definiert. Wir nehmen an, daß ausreichend geklammert ist.

$$\begin{array}{lll}
\psi(\varepsilon) & := & \{\varepsilon\}. \text{ Hier ist rechts der leere String gemeint.} \\
\psi(a) & := & \{a\} \text{ für } a \in \Sigma \\
\psi(rs) & := & \psi(r)\psi(s). \text{ D.h. } = \{xy \mid x \in \psi(r), y \in \psi(s)\}. \\
\psi(r+s) & := & \psi(r) \cup \psi(s). \\
\psi((r)) & := & \psi(r) \\
\psi(r^*) & := & \psi(r)^*.
\end{array}$$

Beachte: für Mengen R von Strings ist R^* definiert durch: zunächst definiert man: R^n für positive Zahlen n als: $R^0 := \{\varepsilon\}$, $R^{n+1} := RR^n$. Danach: $R^* := \bigcup\limits_{n \geq 0} R^n$

Die durch reguläre Ausdrücke bezeichneten Mengen von Worten heißen *reguläre Mengen*.

Die Syntax regulärer Ausdrücke wird oft erweitert durch: R^+, R^n, und $[R]$, wobei $R^+ = RR^*$, $R^n = \underbrace{R \dots R}_{n}$, und $[R] = (R + \varepsilon)$.

Rechenregeln für reguläre Ausdrücke:

$$\begin{array}{lll}
r+s & = & s+r \\
(r+s)+t & = & r+(s+t) \\
r(st) & = & (rs)t \\
r(s+t) & = & rs+rt \\
(s+t)r & = & sr+tr \\
(r+\varepsilon)^* & = & (\varepsilon+r)^* = r^* \\
(r^*)^* & = & r^*
\end{array}$$

Es gibt formale Sprachen, die nicht-regulär sind, z.B. $\{0^n 1^n \mid n \geq 0\}$, aber von einer kontextfreien Grammatik erzeugt werden: $\mathbf{E} ::= \varepsilon \mid 0\mathbf{E}1$

Es gilt (ohne Beweis) : die durch reguläre Ausdrücke definierten formalen Sprachen sind genau die durch reguläre Grammatiken erzeugbaren.

Beispiel 28. für reguläre Ausdrücke und reguläre formale Sprachen.

- $(0+1+2+3+4+5+6+7+8+9)^*$ ist die Menge der Zahldarstellungen.
- $0+(1+2+3+4+5+6+7+8+9)(0+1+2+3+4+5+6+7+8+9)^*$ ist die Menge der Zahldarstellungen ohne führende Nullen.
- Menge der Binärzahlen mit gerader Parität: $(0^*10^*10^*)^*$.
- alphanumerische Strings: "$(a + \ldots + z + 0 + \ldots + 9 + ␣)^*$", wobei ␣ für das blank stehen soll.

3.14.3 Erzeugung einer kontextfreien Grammatik aus einem regulären Ausdruck

Folgende Übersetzung erzeugt aus einem regulären Ausdruck r eine Grammatik G_r, so daß $L(G_r) = \psi(r)$.

$\tau(\varepsilon)$	$\rightarrow$	ε
$\tau(a)$	$\rightarrow$	a
$\tau(rs)$	$\rightarrow$	$\tau(r)\tau(s)$
$\tau((r))$	$\rightarrow$	E; neue Regel $E ::= \tau(r)$, wobei E neues Nichtterminal
$\tau(r+s)$	$\rightarrow$	E; neue Regel $E ::= \tau(r) \mid \tau(s)$, wobei E neues Nichtterminal
$\tau(r*)$	$\rightarrow$	E; neue Regel $E ::= \varepsilon \mid \tau(r)E$, wobei E neues Nichtterminal

Man startet den Algorithmus durch Erzeugen der Regel $\sigma ::= \tau(r)$
Dieser Algorithmus ergibt i.a. keine reguläre Grammatik.

Beispiel 29. Der reguläre Ausdruck sei $a(a+b)^*$:
Die erzeugte Grammatik ist:

σ	::=	aE
E	::=	ε | FE
F	::=	a | b

Man kann diese aber durch eine einfache Transformation in eine reguläre Grammatik umwandeln.

3.15 Endliche Automaten

Um die Frage der maschinellen Verarbeitung von Information, z.B. Erkennen von Mustern, Berechnungen, usw. zu diskutieren, sind in der Informatik eine Reihe abstrakter Maschinenmodelle entwickelt worden. Ein einfaches Modell sind die deterministischen endlichen Automaten.

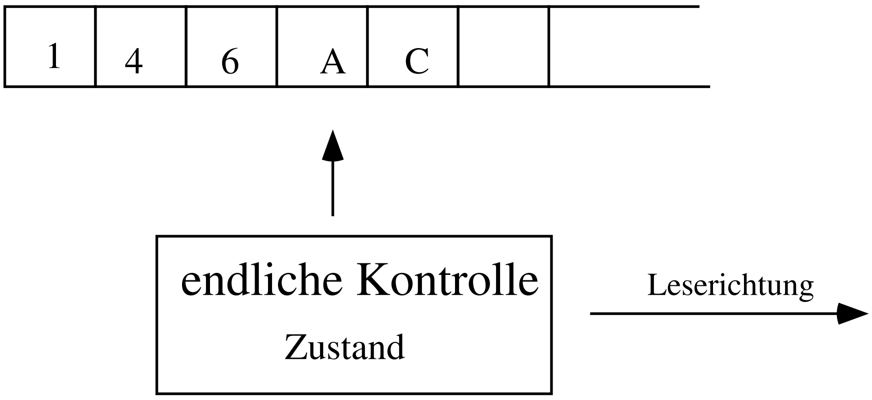
Definition 4. Ein deterministischer *endlicher Automat* A ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei

Q	Menge der Zustände
Σ	Eingabealphabet
δ	(partielle) Übergangsfunktion: $\delta : Q \times \Sigma \rightarrow Q$
q_0	Startzustand
F	akzeptierende Zustände $F \subseteq Q$

Wir erweitern δ zu δ^*: $\delta^*(q, aw) = \delta^*(\delta(a), w)$, $\delta^*(q, a) = \delta(q, a)$, wobei δ^* auf Strings statt auf einzelnen Zeichen operiert, und $\delta^* : Q \times \Sigma^* \rightarrow Q$.

Ein Wort w wird *akzeptiert*, wenn $\delta^*(q_0, w) \in F$.

Man kann einen deterministischen endlichen Automaten darstellen als eine black box, die ein Eingabeband liest und eine endliche Menge von inneren Zuständen hat. Beim Lesen des Eingabebandes wechselt der innere Zustand in Abhängigkeit vom vorhergehenden Zustand und vom gelesenen Zeichen.



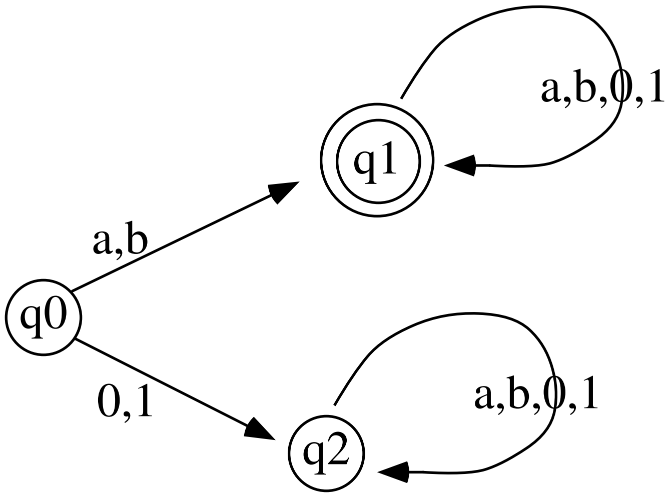
Der Unterschied zur Turingmaschine ist:

- Schreiben auf das Band ist nicht zulässig
- Das Band wird von links nach rechts gelesen

Zum Maschinenmodell des endlichen Automaten gehören die regulären Mengen. Denn die Menge der Worte, die von einem endlichen Automaten akzeptiert wird, ist eine regulären formale Sprache.

Eine graphische Darstellung von endlichen Automaten sind Übergangsdiagramme, die einen deterministischen endlichen Automaten als einen Graph mit markierten gerichteten Knoten und Kanten darstellen. Die Knoten sind mit den Zuständen markiert, die Kanten sind die Übergänge und mit Zeichen des Alphabets markiert.

Beispiel 30. Wir betrachten einen endlichen Automaten für die Erkennung von Bezeichnern über dem Alphabet $\{a, b, 0, 1\}$:



"a10ab"wird akzeptiert, "1ab01"wird nicht akzeptiert.

- $Q = \{q0, q1, q2\}$

- $\Sigma = \{a, b, 0, 1\}$
- $\delta: \quad \delta(q0, a) = q1, \delta(q0, b) = q1$
  $\delta(q0, 0) = q2, \delta(q0, 1) = q2$
  $\delta(q1, x) = q1, \delta(q2, x) = q2$ für $x \in \Sigma$.
- q_0 Startzustand
- $F = \{q1\}$

Dieser DEA ist deterministisch, da von jedem Knoten und für jedes Eingabezeichen nur ein Pfeil ausgeht.

3.15.1 Umwandlung eines DEA in eine reguläre Grammatik

Folgendes Vorgehen erzeugt eine reguläre Grammatik G, so daß $L(G_D) = A(D)$.

- Für jeden Zustand q gibt es ein Nichtterminal E_q.
- Für jeden Übergang $q_1 \xrightarrow{a} q_2$ eine Regel $E_{q_1} ::= aE_{q_2}$. Wenn E_q ein akzeptierender Zustand ist, dann auch die Regel $E_q ::= \varepsilon$.
- Der Startzustand q_0 ergibt das Startsymbol $\sigma = E_{q_0}$ der Grammatik.

Beispiel 31. Betrachte Beispiel 30. Seien E_i die Nichtterminale, die zu q_i gehören. Die zugehörige Grammatik ist:

$$\begin{array}{lll}
\sigma & := & E_0 \\
E_0 & := & (a|b)E_1 \\
E_0 & := & (0|1)E_2 \\
E_1 & ::= & (0|1|a|b)E_1 \\
E_1 & ::= & \varepsilon \\
E_2 & ::= & (0|1|a|b)E_2
\end{array}$$

Für die effiziente lexikalische Analyse und Fehlererkennung benötigt man noch etwas mehr: der Zustand E_2 ist eine Sackgasse, die nie mehr zu einem akzeptierenden Zustand führt. Es ist kein großes Problem, für einen gegebenen DEA die Zustände zu bestimmen die zu Sackgassen gehören. Es ist genau das Problem der Erreichbarkeit eines Knotens in einem gerichteten Graphen.

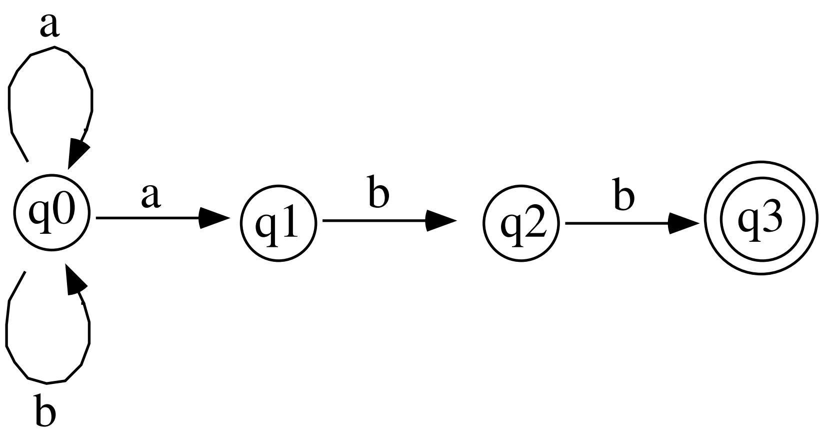
3.15.2 Nichtdeterministische endliche Automaten

Definition 5. Ein *nichtdeterministischer endlicher Automat* (NEA) ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$, wobei

Q	Menge der Zustände
Σ	Eingabealphabet
δ	Übergangsfunktion: $\delta : Q \times \Sigma \to \mathcal{P}(Q)$. Ein Element in $\mathcal{P}(Q)$ entspricht der Menge der möglichen Folgezustände.
q_0	Startzustand
F	akzeptierende Zustände $F \subseteq Q$

Man kann δ zu δ^* erweitern: $\delta^*(q, a) := \delta(q, a)$, $\delta^*(q, aw) := \bigcup\{\delta^*(q', w) \mid q' \in \delta(q, a)$, wobei $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$. Ein Wort w wird akzeptiert, wenn $\delta^*(q_0, w) \cap F \neq \emptyset$.

Beispiel 32. Ein NEA, der $(a+b)^*abb$ erkennt:



Man sieht: von q_0 gehen für a zwei Pfeile aus. Ein Wort wird akzeptiert, wenn es einen Weg zu einem akzeptierenden Zustand gibt.

3.15.3 NEAs mit ε-Übergängen

Bei dieser Erweiterung eines NEA ist es zugelassen, daß in Übergangsdiagrammen Pfeile mit ε markiert sind:

$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$

D.h. es sind Zustandsänderungen erlaubt, **ohne** ein Zeichen der Eingabe zu verbrauchen. Dies ergibt keine wesentliche Erweiterung, denn alle ε-Übergänge kann man leicht eliminieren, indem man für jeden Zustand q die Menge $E(q)$ der Zustände berechnet, die mit (beliebig vielen) ε-Übergängen erreichbar sind.

Jetzt definieren wir zunächst die Funktion δ_ε, die den NEA beschreibt, der alle ε- Übergänge schon durchführt:

$$\delta_\varepsilon(q,a) := \bigcup\{E(p') \mid p' \in \delta(p,a), p \in E(q)\}$$

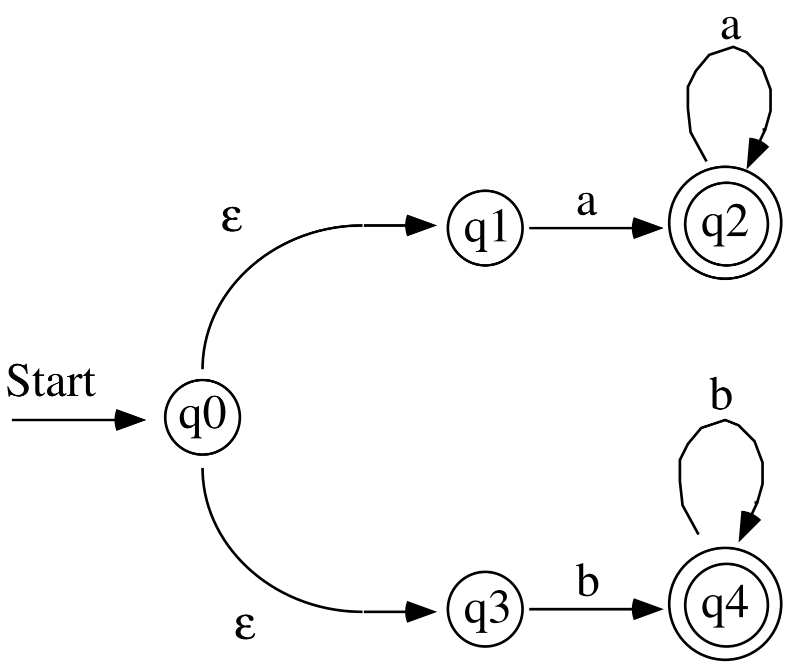
Die zugehörige Funktion δ^* definiert man dann folgendermaßen:

$$\delta^*(q,a) := \delta_\varepsilon(q,a), \delta^*(q,aw) := \bigcup\{\delta^*(p,w) \mid p \in \delta_\varepsilon(q,a)\}$$

Mit diesen Vorbereitungen könnte man die ε-Übergänge eliminieren, indem man folgende Übergänge hinzufügt: Wenn $q_2 \xrightarrow{a} q_3$, $q_2 \in E(q_1)$ und $q_4 \in E(q_3)$, dann $q_1 \xrightarrow{a} q_4$. Wenn dies vollständig durchgeführt wurde, löscht man alle ε-Übergänge.

Der Vorteil der ε-Übergänge liegt in der erleichterten Konstruktion und Spezifikation von NEAs, insbesondere aus regulären Ausdrücken.

Beispiel 33. NEA, der $(aa^* + bb^*)$ erkennt.



3.15.4 Umwandlung eines NEA in einen äquivalenten DEA

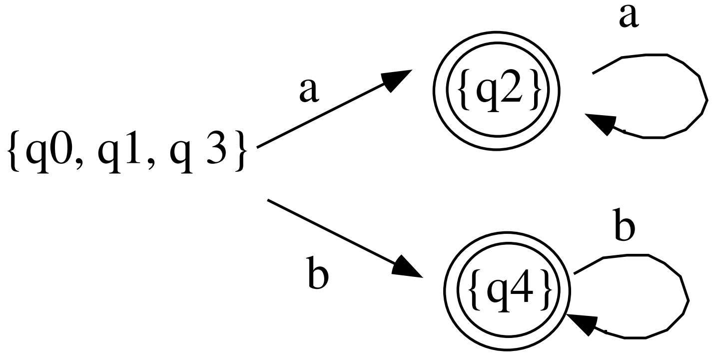
Bei dieser Umwandlung soll die akzeptierte Sprache gleich bleiben.

Gegeben ist ein NEA: $N = (Q, S, \delta, q_0, F)$, daraus konstruieren wir einen äquivalenten DEA D wie folgt:

- Die Zustände von D sind die Teilmengen von Q.
- Die Übergangsfunktion δ_D ist so beschaffen, daß sie alle möglichen Übergänge von N "parallel" simuliert: $\delta_D : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$
  und $\delta_D(M, a) = \bigcup\{\delta_\varepsilon(q, a) \mid q \in M\}$
- Das Startsymbol ist $\{q_0\}$
- Die akzeptierenden Zustände sind: $\{M \subseteq Q \mid M \cap F \neq \emptyset\}$

Beispiel 34. Betrachte das Beispiel 33

Der zugehörige deterministische Automat hat folgendes Übergangsdiagramm:



Problem dieser Umwandlung ist die möglicherweise exponentiell wachsende Anzahl der Zustände.

3.15.5 Transformation einer regulären Grammatik in einen NEA + ε-Übergänge

Diese Transformation zeigt den engen Zusammenhang zwischen regulären Grammatiken und endlichen Automaten. Sie zeigt auch, daß ein NEA mit ε-Übergängen ein geeignetes Zielmodell ist für solche Transformationen.

Man transformiert in 2 Schritten:

Schritt 1: Alle Regeln der regulären Grammatik werden durch Hinzufügen von Nichtterminalen so abgeändert, daß die rechten Seiten von der Form E,, aE, oder ε sind.

Dies ist unproblematisch, indem man z.B. $E ::= abF$ umsetzt in $E ::= aE_1, E_1 ::= bF$, und z.B. $E_3 ::= ab$ in $E_3 ::= aE_4, E_4 ::= bE_5, E_5 ::= \varepsilon$

Schritt 2 Für jedes Nichtterminal E erzeuge einen Zustand q_E. Für die Regeln erzeuge Übergänge nach folgendem Schema:

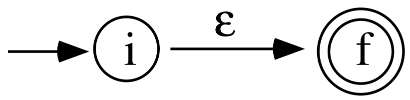
$E_1 ::= E_2 \quad \rightsquigarrow \quad q_{E_1} \xrightarrow{\varepsilon} q_{E_2}$
$E_1 ::= aE_2 \quad \rightsquigarrow \quad q_{E_1} \xrightarrow{a} q_{E_2}$
$E_1 ::= \varepsilon \quad \rightsquigarrow \quad q_{E_1} \in F$, d.h. q_{E_1} wird ein akzeptierender Zustand.

3.15.6 Erzeugung eines NEA aus einem regulären Ausdruck

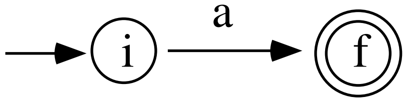
Diese Erzeugung soll zu einem regulären Ausdruck r einen NEA erzeugen, der genau die formale Sprache akzeptiert, die dem regulären Ausdruck entspricht. Wir geben die formale Definition mittels Bildern an.

Es ist jeweils so, daß mit i ein neuer Anfangszustand und mit f eine neuer Endzustand bezeichnet wird. Die Erzeugung ist rekursiv und benötigt, daß die rekursive Konstruktion genau einen Endzustand erzeugt. In den Bildern soll $N(t)$ jeweils die rekursive Konstruktion eines NEA zu einem Unterausdruck sein. Die Anfangs- und Endzustände werden jeweils mit Kreisen angezeigt.

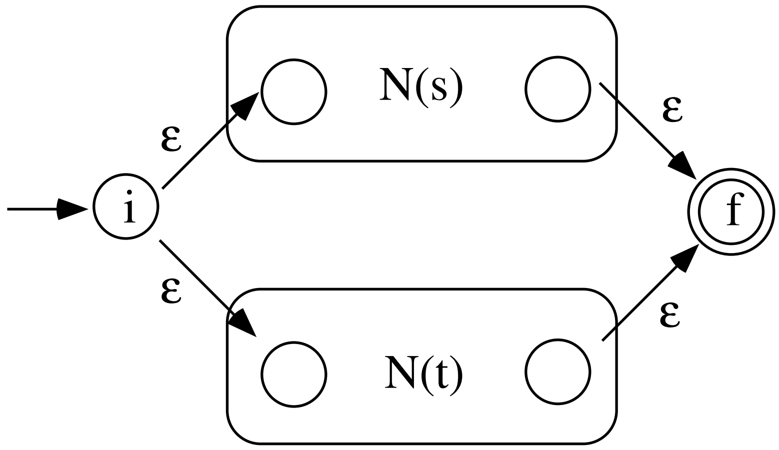
ε:



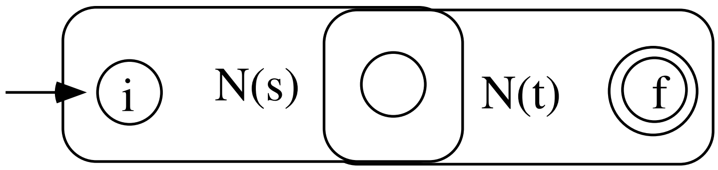
a:



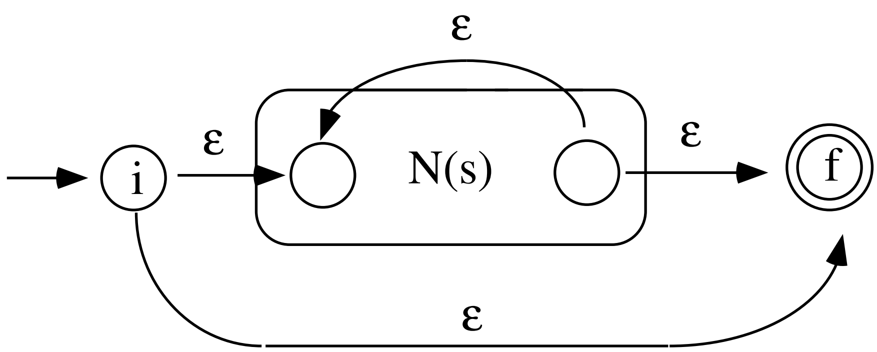
s+t :



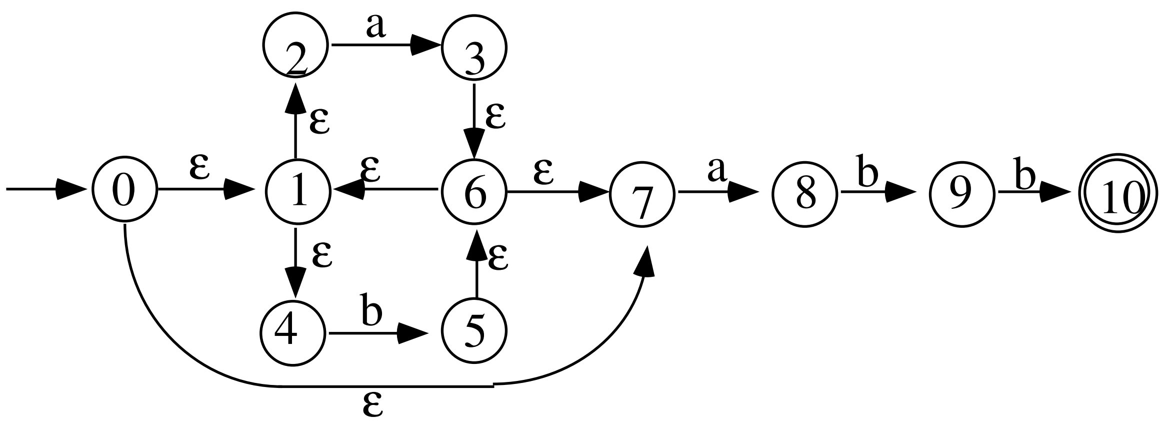
(st)



s*



Beispiel 35. Für die Konstruktion eines NEA aus einem regulären Ausdruck $(a+b)^*abb$:

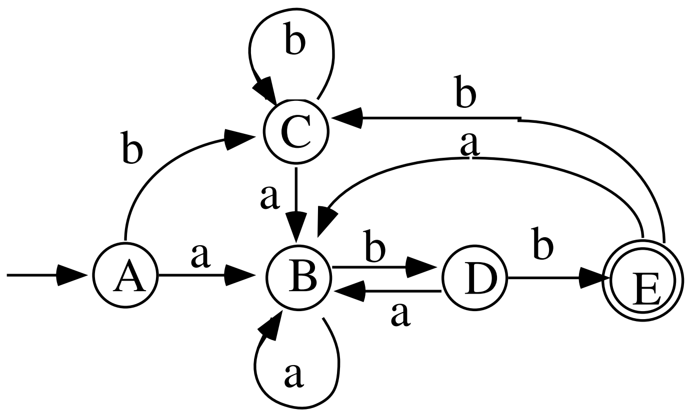


Hieraus kann man einen DEA konstruieren:

Wir benennen nur die Zustände, die tatsächlich auftreten: $A = E(0) = \{0, 1, 2, 4, 7\}$

$A \xrightarrow{a} \{3, 8\} \cup \{1, 2, 4, 6, 7\} = \{1, 2, 3, 4, 6, 7, 8\} =: B$
$A \xrightarrow{b} \{5\} \cup \{1, 2, 4, 6, 7\} = \{1, 2, 4, 5, 6, 7\} =: C$
$B \xrightarrow{a} \{1, 2, 3, 4, 6, 7, 8\} = B$
$B \xrightarrow{b} \{5, 9\} \cup \{1, 2, 4, 6, 7\} = \{1, 2, 4, 5, 6, 7, 9\} =: D$
$C \xrightarrow{a} B$
$C \xrightarrow{b} C$
$D \xrightarrow{a} \{3, 8\} \cup \{1, 2, 4, 6, 7\} = B$
$D \xrightarrow{b} \{5, 10\} \cup \{1, 2, 4, 6, 7\} = \{1, 2, 4, 5, 6, 7, 10\} =: E$
$E \xrightarrow{a} B$
$E \xrightarrow{b} C$

Der zugehörige DEA kann folgendermaßen gezeichnet werden:



3.15.7 Implementierung von DEA, NEA

Wenn man endliche Automaten zum Scannen verwenden will, dann muß man die Übergangsfunktion δ und δ^* des DEA implementieren. Die Eingabeargument sind: der String und der Zustand; die Ausgabe ist der neue Zustand. Die Funktion δ^* hat als Anfangszustand den Startzustand und verarbeitet den ganzen eingegeben String.

Stellt man sich den Automaten als Kiste vor, die ein Eingabeband liest und bei der ein grünes Licht angeht für "akzeptiert", ein rotes für "nicht akzeptiert", so geht beim Lesen des Eingabebandes je nach innerem Zustand das rote oder das grüne Licht an. Es ist auch möglich, daß nach dem grünen Licht wieder rot kommt.

Zum Scannen eines Bezeichners in einem String wie "Ax1 123" wird also das grüne Licht angehen, bei den ersten drei Zeichen angeschaltet bleiben und beim blank wieder ausgehen.

Man erkennt:

- Man benötigt eine leichte Abwandlung des Automaten, der erst beim blank den Bezeichner erkennen und akzeptieren soll (ohne das blank).

- Der Automat braucht eigentlich verschiedene akzeptierende Zustände, wenn er auch die Zahl danach erkennen soll.

- Es wird nicht nur der akzeptierende Zustand als Ausgabe benötigt, sondern auch der String, der zum Bezeichner gehört, bzw. die Zahl die zu "123" gehört. D.h. der Zustand alleine reicht als Ausgabe nicht aus: es muß eine andere Ausgabemöglichkeit geben.

Verwendet man NEAs bei der Implementierung, so kann man, statt mit dem Zustand, mit der Potenzmenge der Zustände die Berechnung durchführen. Dies gilt auch für die Erweiterung mit ε-Übergängen.

Diese Implementierung realisiert den NEA, wobei die Zeit zur Berechnung der Übergangsfunktion für jedes eingegebene Zeichen polynomialer in Abhängigkeit von der Größe des NEA ist: Es sind Mengenoperationen erforderlich: Vereingungs- und Elementtests, deren Aufwand nur vom NEA abhängt.

3.15.8 Zusammenfassende Bemerkungen

- jede reguläre Menge kann durch einen endlichen Automaten erkannt werden.

- die von endlichen deterministischen und nicht-deterministischen Automaten erkannten Mengen sind genau die regulären Mengen.

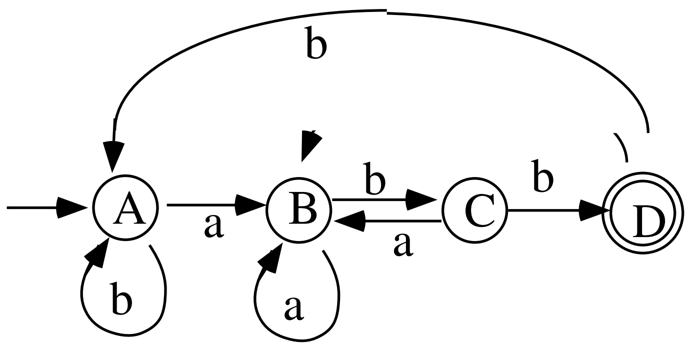
- jeder nicht-deterministische Automat kann in einen deterministischen umgewandelt werden, der die gleiche Sprache akzeptiert.

- Es gibt eine Reihe wichtiger Anwendungen für nicht-deterministische Automaten, z.B. haben diese oft weniger Zustände als die deterministische Variante und sind daher einfacher zu spezifizieren.

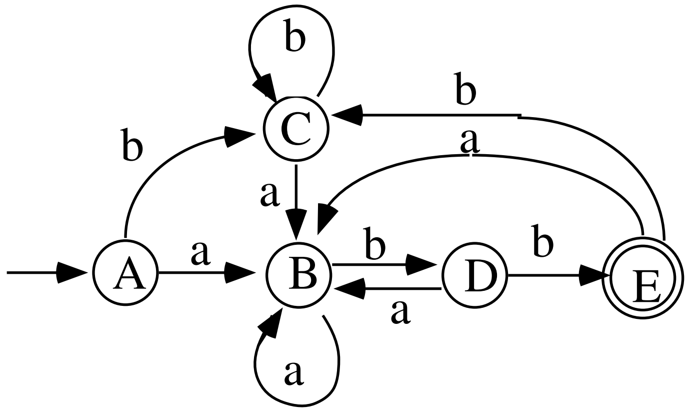
3.15.9 Minimierung von DEAs

Die folgenden beiden DEA erkennen beide (a+b)*abb.

N1:



N2:



Kann man N2 verkleinern (minimieren) durch Reduktion der Anzahl der Zustände?

Das ist möglich:

Idee: Berechne Äquivalenzklassen von Zuständen mit gleichem Akzeptanzverhalten.

Algorithmus: Zerlege sukzessive Mengen von Zuständen in Untermengen, die sich bzgl. des Verhaltens, bezogen auf die aktuelle Menge von Mengen, unterscheiden. Starte dazu mit der Zerlegung in zwei Mengen: i) akzeptierende Zustände und ii) andere Zustände.

1. $\{A, B, C, D\}\{E\}$ Anfangszerlegung.
2. $\{A, B, C\}, \{D\}, \{E\}$ (nur D hat b-Übergang nach E)
3. $\{A, C\}, \{B\}, \{D\}, \{E\}$ (nur B hat b-Übergang nach D)
4. Danach stabilisiert sich diese Zerlegung.

Der minimierter DEA ist äquivalent zu N1.

Es gilt:

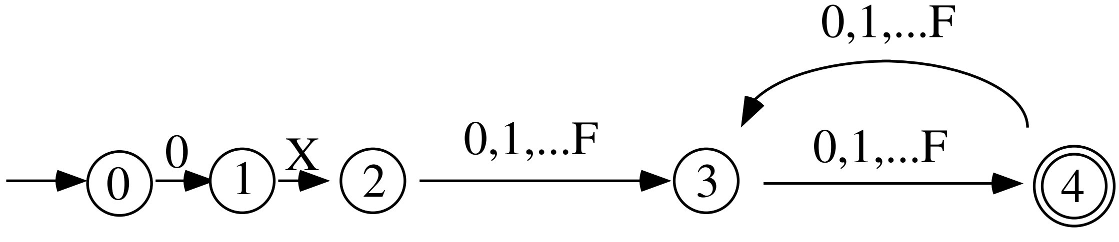
Satz Myhill-Nerode *Für eine reguläre Sprache L gibt es (bis auf Umbenennung der Zustände) genau einen minimalen DEA.*

Beispiel 36. Hexadezimalzahlen mit der Syntax $0X$**HexSym**...**HexSym**, und **HexSym** ::= 0 | ... | 9 | A ... | F

Der reguläre Ausdruck ist:

`0'X'`(0 + 1 $\ldots verb!'F'!$) (0 + 1 $\ldots verb!'F'!)^*$.

Ein DEA dazu ist:



Die Zustände 1, 2, und 3 sind Fehlerzustände.

Man kann auch noch Leerzeichen am Anfang oder Ende zulassen. Dieser DEA kann nur erkennen, ob der eingegebene String eine HexaDezimalZahl mit Vorzeichen und Exponent ist. Es fehlt die Ausgabe der erkannten Zahl, des "Wertes".

3.15.10 Mealy-Automat

Zur Formalisierung der Ausgabe gibt es ein formale Erweiterung eines DEA:

Definition 6. Ein endlicher Automat M vom *Mealy-Typ* ist ein 6-Tupel

$$M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$$

Hierbei ist Q, Σ, δ, q_0 wie bei einem endlichen Automaten. Allerdings gibt es keine akzeptierenden Zustände. Zusätzliche Bestandteile sind:

Δ Alphabet der Ausgabezeichen:
λ die *Ausgabefunktion.* $\lambda : Q \times \Sigma \to \Delta$.

Dies ist ein deterministischer endlicher Automat, der bei jedem Eingabezeichen zusätzlich ein Ausgabezeichen druckt, das vom aktuellen Zustand und vom Eingabezeichen abhängig ist. Es ist natürlich möglich, daß Σ und Δ gemeinsame Zeichen haben.

Wenn die Ausgabefunktion nur vom Zustand abhängt, spricht man auch von *Moore-Automaten.*

Allerdings ist es beim Programmieren z.B. mit Haskell kein Problem, das Werkzeug des DEA mit einer Ausgabefunktion zusammen zu verwenden.

und Ermittlung des Wertes: