Kapitel 3

Übersetzer (Compiler): Von der Programmiersprache zur Maschinensprache

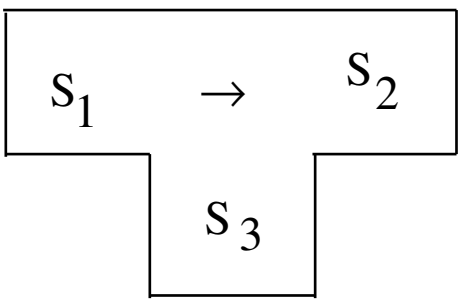
3.1 Compiler

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 ⟨ Ziffer ⟩ 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|...|9

Statt der drei Regeln können wir auch schreiben:

Ausdruck ::= **Ausdruck** + **Ziffer**| **Ausdruck** − **Ziffer**|**Ziffer**
Ziffer ::= 0|...|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.

Ausdruck ::= **Ausdruck** + **Ziffer** | **Ausdruck** − **Ziffer**|**Ziffer**
Ziffer ::= 0|...|9

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

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

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

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 Ableitung 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.

A	::=	**A** + **A**
A	::=	**A** − **A**
A	::=	$0 \mid \ldots \mid 9$

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

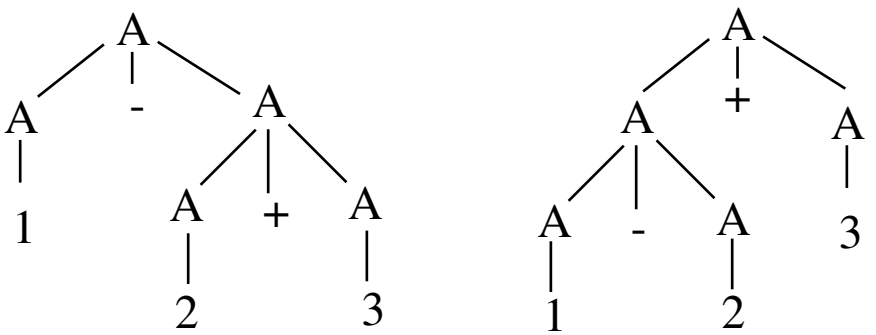
Es gilt:

- 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.

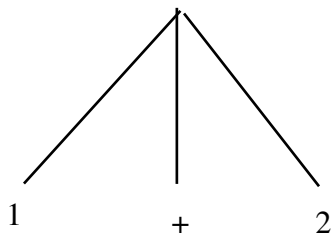
Beispiel 8. Beispielgrammatik, die Herleitungsbäume und die Mehrdeutigkeit demonstriert.

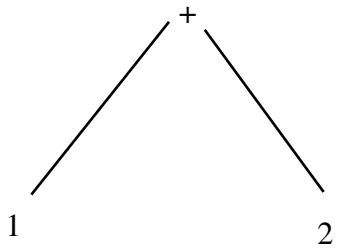
$\mathbf{A} \quad ::= \quad \mathbf{A} + \mathbf{A}$
$\mathbf{A} \quad ::= \quad \mathbf{A} - \mathbf{A}$
$\mathbf{A} \quad ::= \quad 0 \mid \ldots \mid 9$

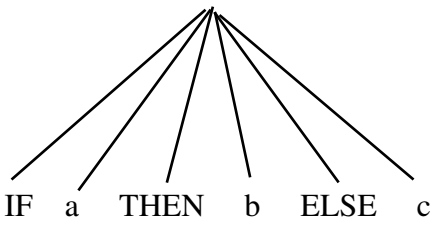
Zwei verschiedene Herleitungsbäume für $1 - 2 + 3$ sind:

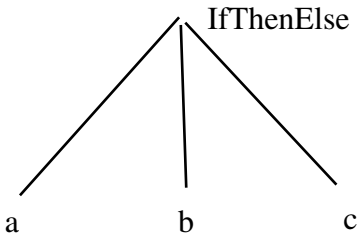


Statt Herleitungsbäumen verwendet man auch eine kompakte Darstellungsweise (des Baumes, die von der genauen Grammatik absieht, sogenannte Syntaxbäume. Im allgemeinen geht das Verfahren so vor: 1) Knoten werden mit dem Namen der Regel markiert, wobei auch Operatornamen wie $+, -, *$ verwendet werden. 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:

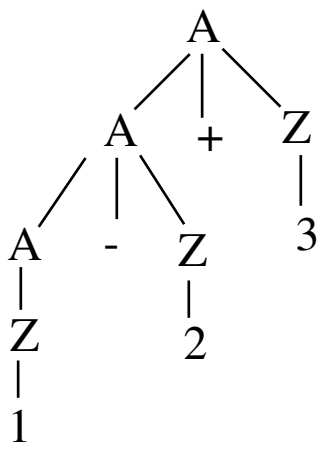
Definition 2. Eine Grammatik G heißt *eindeutig*, wenn es für jedes Wort der zugehörigen Sprache $L(G)$ genau einen Herleitungsbaum gibt.

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.

S	::=	**AB**
A	::=	0 | 1
B	::=	8 | 9

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.

S ::= **A** |**B**
A ::= 0**A** |1
B ::= 0**B** | 2

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 Mange 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 FIRST(w_1) disjunkt zu 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 angewednet 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:

Expr ::= **Term Rest**
Rest ::= + **Term Rest** | − **Term Rest**| ε
Term ::= 0 | ... | 9

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 Produktion zu eliminiern, deren rechte Seite ein ε ist. Das Problem dieser Produktionen ist, daß sie (bei 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). 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.

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).

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

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

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:

A	::=	β**B**
B	::=	α **B** | ε

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: