Kapitel 5

Compiler: Grundlagen

Unterlagen zu Programmierung 2 im Sommersemester 2017

5.1 Compiler, Übersetzer

Eine etwas vereinfachte Definition ist:

> Ein Übersetzer (Compiler) ist ein Programm, das ein Wort einer formalen Sprache S_1 (den Quelltext) in ein Wort einer anderen formalen Sprache S_2 (den Zieltext) umwandelt.

Beispiele sind Programme in einer Programmiersprache wie Haskell, Java, Python, oder PASCAL, die in ein Maschinenprogramm oder in Byte-code übersetzt werden, d.h. in eine andere formale Sprache, aber auch LaTex-Eingaben, die in eine PostScript oder PDF-Datei übersetzt werden.

Der Compiler selbst ist ein Programm, das möglicherweise in einer weiteren Programmiersprache S_3 geschrieben ist.
Eine korrektere Beschreibung ist:

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

Eingeschränkt auf Programmiersprachen:

> Ein Compiler ist ein Programm, das ein Programm einer Programmiersprache S_1 (das Quellprogramm) in ein Programm einer anderen Programmiersprache S_2 umwandelt (das Zielprogramm), wobei die Semantik erhalten bleibt

Semantik ist die Bedeutung der Programme. I.a. ist es ausreichend, eine operationale Semantik zu spezifizieren. Das Quellprogramm muss dann das gleiche Verhalten haben wir das Zielprogramm.

Typische Anwendungsfälle für Compiler:

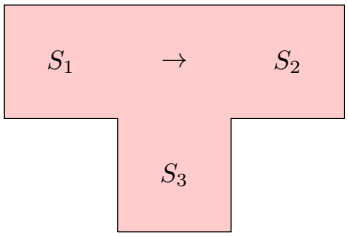
- Übersetzung eines Programms in einer Programmiersprache in ein Programm in einer Assemblersprache, so dass das Assemblerprogramm das richtige tut.
- Ein Compiler, der S_1-Programme in S_2-Programme übersetzt, damit ein weiterer Übersetzer von S_2 nach Assembler verwendet werden kann.

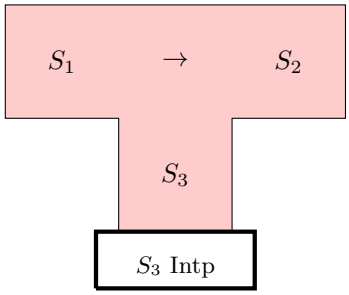
Ein Interpreter ist ein Programm, das den Text eines Programms einliest und dann ausführt, i.a. ohne ein Programm in einer anderen Sprache zu erzeugen.
Hat man die operationale Semantik einer Programmiersprache vollständig gegeben, dann kann man einen

Interpreter schreiben. Genaugenommen ist eine operationale Semantik genau die Spezifikation eines Interpreters.

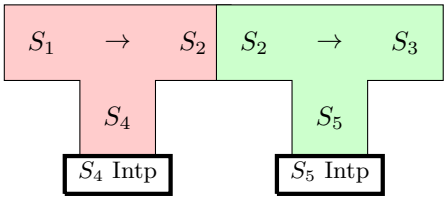
Allerdings ist dies von der Vorgehensweise, zuerst das Programm zu kompilieren, und dann auszuführen, nicht zu unterscheiden. Jedenfalls dann nicht, wenn die operationale Semantik vom Interpreter und Compiler eingehalten wird.
In der Praxis ist allerdings ein interpretiertes Programm meist langsamer in der Ausführung.

Folgendes Puzzleteil (sogenannte T-Diagramme, siehe Wirth: Compilerbau) kann man verwenden, um aufbauend auf Compilern und vorhandenen Interpretern weitere Compiler zu konstruieren, ohne neue schreiben zu müssen. Das Diagramm soll bedeuten, dass ein Programm der Sprache S_1 in ein Programm der Sprache S_2 übersetzt wird, und dass der Übersetzer in der Sprache S_3 geschrieben ist. Im Diagramm dient die Box mit der Kennzeichnung Intp dazu, anzudeuten, dass man Programme der Sprache S_3 ausführen (interpretieren) kann.

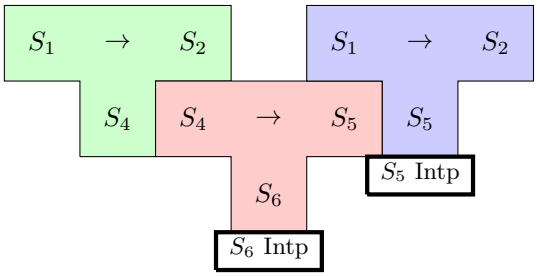




S_1, S_2, S_3 brauchen nicht notwendig verschieden voneinander zu sein. Eine erste Möglichkeit, Compiler zusammenzusetzen, ist die Übersetzung zuerst von S_1 nach S_2 und dann von S_2 nach S_3 wobei im Extremfall zwei verschiedene Interpreter benötigt werden.



Ein zweite Möglichkeit für das Zusammensetzen zweier Compiler und Interpreter zeigt das folgende Bild, wobei das rechteste T-Diagramm den resultierenden Compiler darstellt.



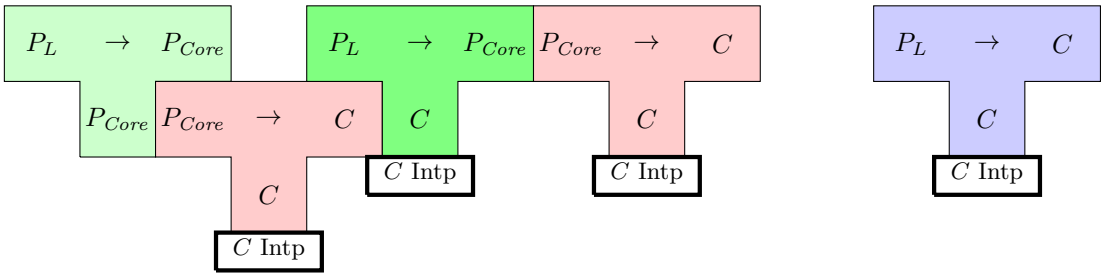
Hier hat man nur Interpreter für S_5, S_6, einen S_4-Übersetzer von S_1 nach S_2, und einen S_6-Übersetzer von S_4 nach S_5, aber keinen Interpreter für S_4. Man kann daraus einen lauffähigen Übersetzer von S_1 nach S_2 konstruieren, der nur noch auf einen S_5-Interpreter angewiesen ist.

Diese Problematik tritt auf, wenn man eine eigene Programmiersprache konstruieren will, und den zugehörigen Compiler möglichst schon in dieser Programmiersprache schreiben will. Normalerweise schreibt man dann einen Interpreter/Compiler für einen möglichst kleinen Ausschnitt dieser Sprache in einer schon vorhandenen Programmiersprache, und schreibt dann Module unter Benutzung diesen Ausschnitts, und setzt dann den Compiler aus weiteren solchen Teilen zusammen. (Bootstrapping: wie man sich selbst aus dem

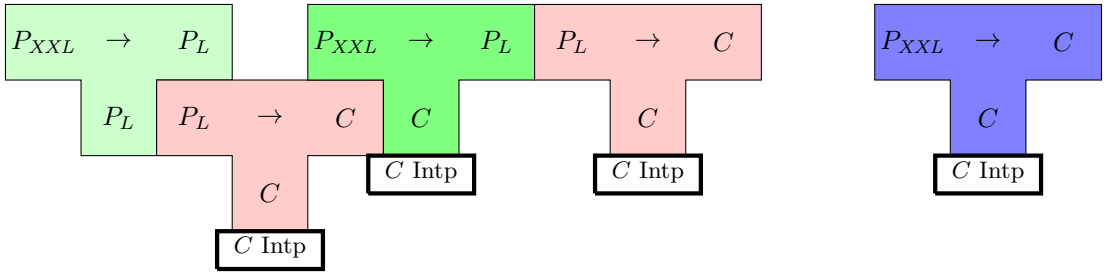
Sumpf zieht)

Der erste Schritt besteht darin, einen Compiler für einen möglichst kleinen Ausschnitt P_{COR} in einer gängigen Programmiersprache zu schreiben (Sagen wir mal C).
Danach schreibt man einen Compiler (bzw. eine Transformation) von $P_L \to P_{COR}$, geschrieben in P_{COR}, und danach einen Compiler (bzw. Transformation) von $P_{XXL} \to P_L$, geschrieben in P_L.

Die folgenden Diagramme zeigen, wie man diese zusammensetzen kann zu einem lauffähigen Compiler für P_{XXL} in C.



Darauf aufbauend, kann man weiter konstruieren:



Die Syntax, d.h. die Beschreibung der möglichen (grammatikalisch korrekten) Quellprogramme ist z.B. durch eine BNF oder eine kontextfreie Grammatik definiert.

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. Programme werden hierbei meist als (partielle) Funktionen vom Zustand vorher auf Zustand danach definiert. Hier werden meist vollständige partielle Ordnungen verwendet, die es erlauben, z.B. einer While-Schleife als Objekt die kleinste obere Schranke einer Folge zuzuweisen (Fixpunkt)

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

transformationelle Semantik Die Bedeutung wird definiert durch Transformationsregeln in andere Programmkonstrukte. Diese Methode 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, Python, 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.

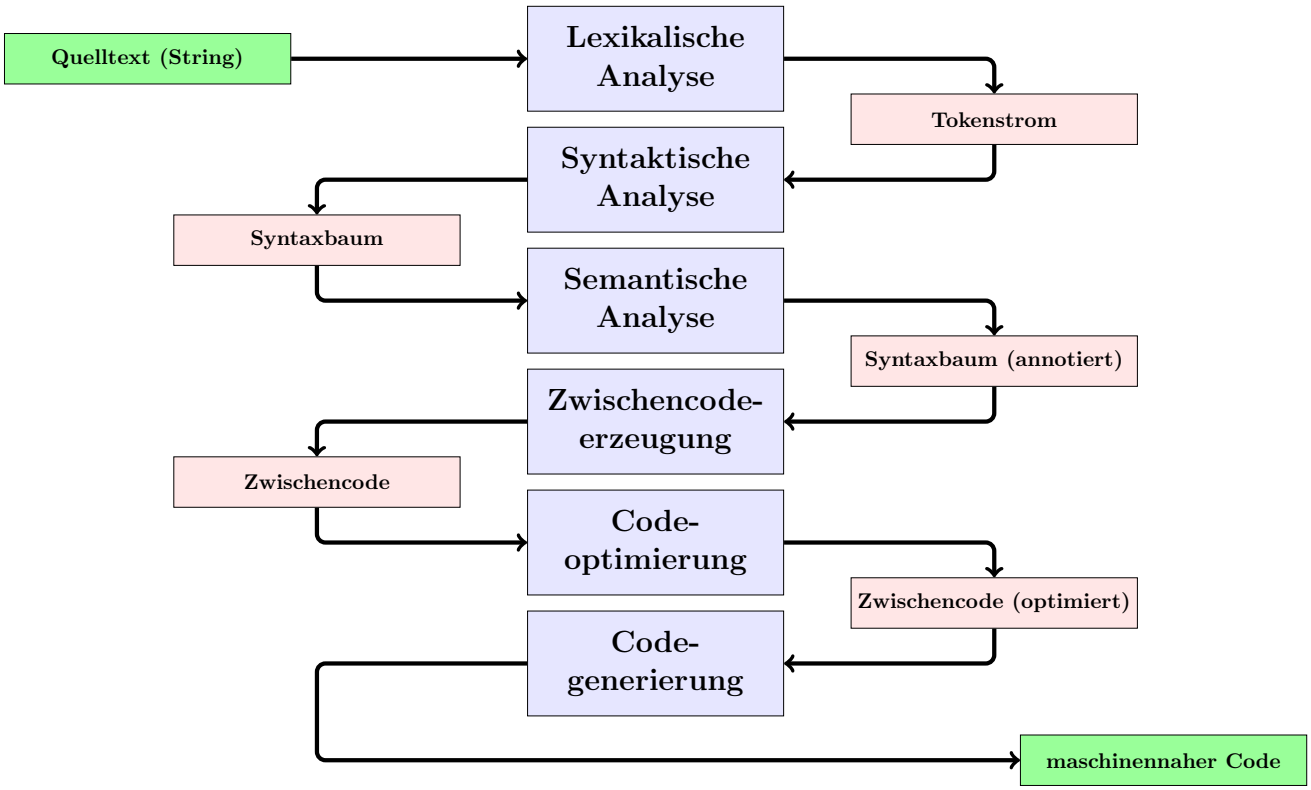
Das Resultat der Assemblierung ist eine Folge codierter Maschinenbefehle, d.h. eine Folge von binären Speicherworten, die direkt vom Prozessor verstanden werden (d.h. ausgeführt werden können). Allerdings ist diese Folge für einen Prozessor anderer Bauart nicht verständlich, bzw. bedeutet etwas anderes.

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); sogenannte Emulation.

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

- Textverarbeitungssprachen, z.B. die Sprache von TeX bzw. LaTeX.
- 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.

5.2 Phasen eines Compilers

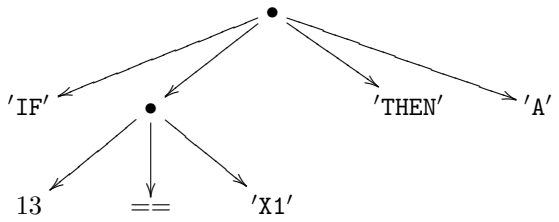


Lexikalische Analyse (Scanning): ein Programm wird als Zeichenkette linear abgearbeitet und in Symbole (tokens) aufgeteilt

Z.B. „IF 13 == X1 THEN A “ wird zu: (’IF’, 13, ==, ’X1’, ’THEN’,’A’)

Syntaxanalyse(parsing): Die Folge von Symbolen wird geprüft, ob sie der zugehörigen Grammatik entspricht. Die Folge wird strukturiert und in eine Baumform umgewandelt.

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



Semantische Analyse: Untersuchung auf semantische Fehler.
z.B. Typüberprüfungen : `1 + 'a'` zurückgewiesen. Ebenso findet hier die Kontext-überprüfung: Überprüfung der Gültigkeitsbereiche von Variablen; Sind Deklarationen vorhanden?, usw.

Zwischencode-Erzeugung: Generierung von Code für eine abstrakte Maschine

Codeoptimierung: Verbesserung des Zwischencodes

Code-Erzeugung: Erzeugung eines Programms (i.a. Assembler, aber auch C) 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 im Fehlerfall ermöglichen soll.

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

Einleseroutine: Zeichenbehandlung. Diese ist abhängig vom Prozessor bzw. Betriebssystem, Das wird aber i.a. durch vorgegeben Einleseroutinen und Standards der Zeichenkodierung in standardisierter Weise durchgeführt.

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

Assembler übersetzt das Assemblerprogramm in Objektcode.

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

5.2.1 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, Zwischencode-Erzeugung, Zwischencode-Optimierung).
- Back-End: maschinenspezifischer Teil (z. B. I/O, Codegenerierung)

5.3 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 5.3.1 *$\Sigma = \{x, y\}$. Dann sind $\{\varepsilon\}$, $\{xy, x, y\}$ formale Sprachen über dem Alphabet Σ.*

Beispiel 5.3.2

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 5.3.3
- $\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$, *die Ziffernrepräsentation aller natürlichen Zahlen ist eine Sprache über* Σ.
  111, 7, 3210999, ...
- $\Sigma = \{A,\ldots,Z,a,\ldots,z,0,1,2,3,4,5,6,7,8,9,),(,\ldots\}$.
  Dann ist der Quell-Code als String aller möglichen Haskell-Programme eine formale Sprache über Σ.

Welche Möglichkeiten gibt es solche formalen Sprachen zu beschreiben? U.a.:

- 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.

5.3.1 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:

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

5.4 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 Programm, das testet, ob ein (langer) String zu einem vorgegebenen regulären Ausdruck gehört und dabei gleichzeitig den String aufteilt, wobei die kleinsten erkannten (akzeptierten) Teilstrings jeweils abgetrennt werden. Praktisch sinnvoll ist das Vorgehen nur, wenn diese Aufteilung eindeutig ist.
Ein Scanner liest ein Programm als lineare Zeichenkette ein und beseitigt z.B. Kommentare sowie Leer- und Tabulatorzeichen. Bestimmte Ketten von Zeichen werden zu *Token* zusammengefasst. Ein solches Token kann ein komplexere Datenstruktur sein, bestehend aus:

- Markierung: Nur diese wird von der Grammatik des anschließenden Parsers erfasst.
- Attribute: wie z.B. Texteingabe, Zahlwert, Position im Eingabefile. 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:

$$\text{Parser} \quad \begin{cases} \texttt{E} & ::= & \texttt{E} + \texttt{T} \mid \texttt{T} \\ \texttt{T} & ::= & \texttt{T} * \texttt{F} \mid \texttt{F} \\ \texttt{F} & ::= & (\texttt{E}) \mid \texttt{Zahl} \mid \texttt{Bezeichner} \end{cases}$$

$$\text{Tokenizer} \quad \begin{cases} \texttt{Zahl} & ::= & \texttt{Ziffer} \mid \texttt{ZifferZahl} \\ \texttt{Ziffer} & ::= & 0|1|2|3|4|5|6|7|8|9 \\ \texttt{Bezeichner} & ::= & \texttt{Char} \mid \texttt{Char RestBez} \\ \texttt{RestBez} & ::= & \texttt{Alphanum RestBez} \mid \texttt{Alphanum} \\ \texttt{Alphanum} & ::= & \texttt{Ziffer} \mid \texttt{Char} \end{cases}$$

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. tatsächliches 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

5.4.1 Fehlererkennung des Scanners

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

- bei der Analyse 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.

5.4.2 Scannen von Kommentaren und Strings

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

Eine bekannte Problematik 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 und 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 −− ode //.

geklammerte Kommentare: nicht geschachtelt sind von Kommentar-Begrenzern eingeschlossen. Z.B. Anfang: /∗ , Ende: ∗/.

geklammerte Kommentare (auch geschachtelt) sind von Kommentar-Begrenzern eingeschlossen. Z.B. Anfang: /∗ , Ende: ∗/. Diese Kommentare können geschachtelt vorkommen. Schachtelung von Kommentaren ist sinnvoll, wenn man größere Teile des Programms auskommentieren will, und kurz danach die Auskommentierung zurücknehmen will. Dies wird jedoch in aktuellen Programmiersprachen nicht im Compiler erledigt, d.h. ist nicht Teil der Programmiersprachensyntax, sondern wird von Editoren durch Zeilenkommentarhandhabung unterstützt.

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

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

5.5 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*, wobei $N \cap T = \emptyset$.
3. $P \subseteq N \times (N \cup T)^*$ eine endliche Menge (Produktionensystem)
4. $\sigma \in N$ ist ein ausgezeichnetes Hilfszeichen *(Startzeichen, Axiom)*

$(A, w) \in P$ 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.

Allgemeine Grammatiken kann man hinschreiben, indem man beliebige Strings in $(N \cup T)^*$ als linke Seite einer Regel erlaubt. Diese sind dann nicht mehr kontextfrei. Man nennt diese *unbeschränkte Grammatik bzw. Semi-Thue-Systeme*. Man nennt eine unbeschränkte 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 5.5.1 *Für kontextfreie Grammatik „Ausdruck“. Die Schreibweise ist eine EBNF (extended BNF).*

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	*::=*	$0 \mid \ldots \mid 9$

5.5.1 Herleitungen

Der Zusammenhang zwischen Grammatik und der durch sie festgelegten Sprache ergibt sich durch den Begriff der *Herleitung*. Informell sind genau diejenigen Worte in der zur Grammatik gehörenden formalen Sprache, die sich durch mehrfache Ersetzung von Nichtterminalen, entsprechend einer der Grammatikregeln, herstellen lassen, wobei man mit dem Startsymbol beginnt. Etwas formaler:

Definition 5.5.2 *Sei $G = (N, T, P, \sigma)$ eine CFG.*

- *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 $w_{i-1} \to w_i$ für $2 \leq i \leq n$.*

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

$$L(G) := \{w \in T^* \mid \sigma \to^* w\}$$

Beispiel 5.5.3

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

Das Startsymbol der Grammatik ist **Ausdruck**, *Terminalsymbole sind* $\{0,\ldots,9,+,-\}$.
Eine Herleitung ist z.B.: **Ausdruck** $\rightarrow$ **Ausdruck+Ziffer** $\rightarrow$ **Ziffer+Ziffer** $\rightarrow$ *1*+**Ziffer** $\rightarrow$ *1*+*2*.
D.h. **Ausdruck** $\rightarrow^*$ *1*+*2*.

5.5.2 Rechts- und Linksherleitungen

Da es verschiedene Herleitungen eines Wortes w aus einer CFG G geben kann, betrachtet man Herleitungen mit festen Strategien. Zwei mit praktischer Bedeutung sind die Rechts- und Linksherleitungen.
Ersetzt man bei einer Herleitung immer nur das am weitesten links stehende Nichtterminal, so ergibt sich eine **Linksherleitung**.

Ersetzt man immer nur das am weitesten rechts stehende Nichtterminal, ist die Herleitung eine **Rechtsherleitung**.

Beispiel 5.5.4 *Sei G wie in Beispiel 5.5.3. Das Startsymbol der Grammatik ist* **Ausdruck**, *Terminalsymbole sind* $\{0,\ldots,9,+,-\}$.
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 (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 5.5.5

A *::=* **A+A**
A *::=* **A - A**
A *::=* *0* | ... | *9*

Rechtsherleitungen *von* `1-2+3`*:*
A $\rightarrow$ **A**-$\underline{\textbf{A}}$ $\rightarrow$ **A-A+** $\underline{\textbf{A}}$ $\rightarrow$ **A-**$\underline{\textbf{A}}$+*3* $\rightarrow$ $\underline{\textbf{A}}$-*2*+*3* $\rightarrow$ `1-2+3`.
Dies würde dem Ausdruck `(1-(2+3))` *entsprechen.*
A $\rightarrow$ **A+**$\underline{\textbf{A}}$ $\rightarrow$ $\underline{\textbf{A}}$+*3* $\rightarrow$ **A-** $\underline{\textbf{A}}$+*3* $\rightarrow$ $\underline{\textbf{A}}$-*2*+*3* $\rightarrow$ `1-2+3`.
Das entspricht `((1-2)+3)`.

Linksherleitungen von `1-2+3`*:*
A $\rightarrow$ $\underline{\textbf{A}}$+**A** $\rightarrow$ $\underline{\textbf{A}}$-**A+A** $\rightarrow$ *1*- $\underline{\textbf{A}}$+**A** $\rightarrow$ *1*-*2*+$\underline{\textbf{A}}$ $\rightarrow$ `1-2+3`.
Das entspricht `(1-2)+3)`.
A $\rightarrow$ $\underline{\textbf{A}}$-**A** $\rightarrow$ *1*-**A** $\rightarrow$ *1*-$\underline{\textbf{A}}$+**A** $\rightarrow$ *1*-*2*+$\underline{\textbf{A}}$ $\rightarrow$ `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.

5.6 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 folgendes gilt:

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

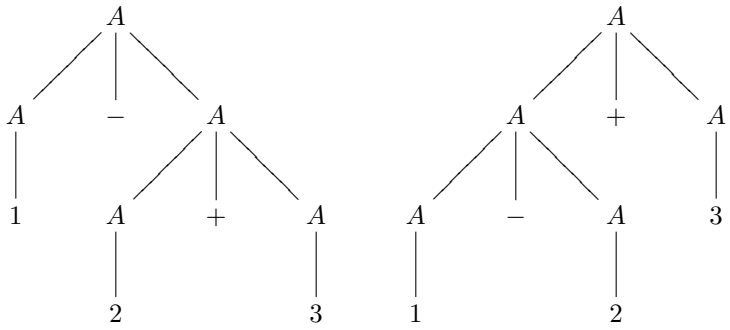
Es gilt:

- zu jeder Herleitung eines Wortes gibt es einen zugehörigen Herleitungsbaum.
- Ein Herleitungsbaum fasst i.a. 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.

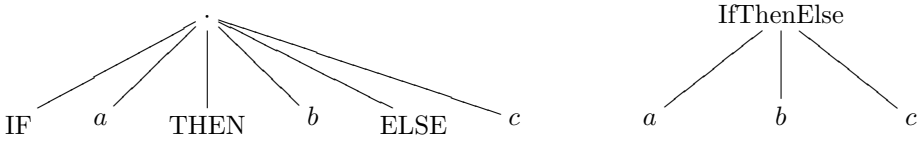
Definition 5.6.1 *Eine Grammatik G heißt* eindeutig*, wenn es für jedes Wort der zugehörigen Sprache $L(G)$ genau einen Herleitungsbaum gibt. Andernfalls heißt die Grammatik* mehrdeutig*.*

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

Beispiel 5.6.2 *Beispielgrammatik, die Herleitungsbäume und die Mehrdeutigkeit demonstriert.*
A ::= **A+A**
A ::= **A-A**
A ::= *0* | ... | *9*
Zwei verschiedene Herleitungsbäume für `1 - 2 + 3` *sind:*



Statt Herleitungsbäumen verwendet man auch eine kompakte Darstellungsweise, 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:



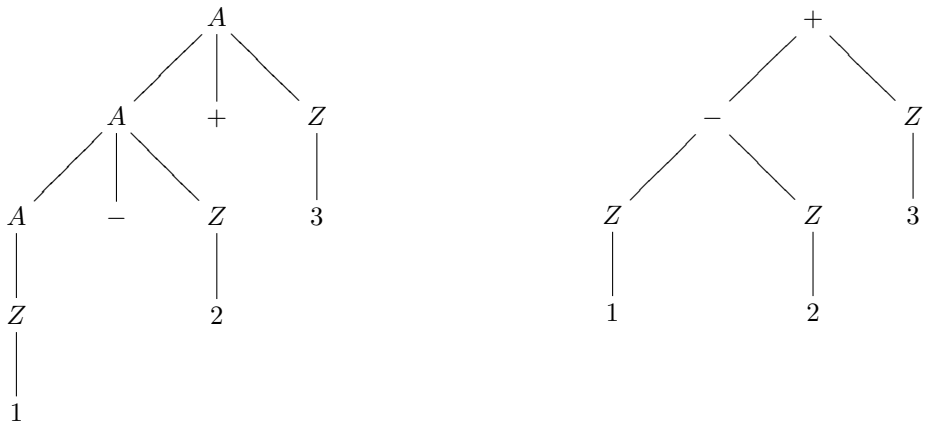
Eine Festlegung der Semantik der Worte einer formalen Sprache bzw. eines Programms einer Programmiersprache, die durch eine Grammatik gegeben sind, wird i.a. über den Herleitungsbaum definiert. D.h. man ordnet jedem Herleitungsbaum eine Bedeutung zu. Obiges Beispiel erlaubt zwei Herleitungsbäume für 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, allerdings nicht operational gleich, sondern nur das operationale Ergebnis[1]. Im allgemeinen muss man jedoch Mittel und Wege finden, um für bestimmte Ausdrücke Eindeutigkeit zu erzielen. Dazu kann man den Grammatikbegriff einschränken:

Beispiel 5.6.3 *Beispiel einer eindeutigen Grammatik mit der gleichen erzeugten Sprache wie Beispiel 5.6.2*

A ::= **A+Z**
A ::= **A-Z**
A ::= **Z**
Z ::= *0* | ... | *9*

Der einzige (d.h eindeutige) Herleitungsbaum für $1-2+3$ *ist, und ein zugehöriger Syntaxbaum 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:* $m_1 op_1 m_2 \dots op_n m_n$ *mit* $m_i \in \{0, \dots, 9\}, 1 \leq n$. *Man sieht dann auch, dass es nur eine Rechtsherleitung für jedes solche Wort geben kann.*

5.7 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

[1] Das gilt nicht für Gleitkommazahlen

Beispiel 5.7.1

Gegeben: *Syntax einer Programmiersprache und der Quelltext eines Programms.*
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, (CYK: Cocke, Younger, Kasami, falls Grammatik in Chomsky-Normalform, oder Earley-Algorithmus) der asymptotische Laufzeit $O(n^3)$ in der Anzahl n der Eingabesymbole hat. Der CYK-Algorithmus benutzt dynamisches Programmieren und erzeugt ein Tabelle, in der für jedes Paar (N, w) von Nichtterminal N und Wort w eingetragen wird, ob $N \rightarrow_G^* w$ gilt oder nicht, angefangen wird mit den kürzesten Worten w. Wenn man w auf Unterstrings des Eingabewortes beschränkt, wird der Algorithmus polynomiell in der Länge des eingegebenen Strings.

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 und die den Eingabestring von links nach rechts abarbeiten. Normalerweise ist die Eingabe ein Strom (ein File, eine Liste) von Zeichen (oder Token), 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: Es wird versucht eine Herleitung vorwärts, vom Startsymbol aus, zu bilden („forward-chaining“)

Bottom-Up: Es wird versucht 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

I.a. wird das Top-Down-Verfahren verwendet, um eine Linksherleitung zu konstruieren, und das Bottom-Up-Verfahren, um eine Rechtsherleitung zu konstruieren.

Beispiel 5.7.2

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 zu parsenden String als Steuerung mitverwenden, um unsinnige Suche auszuschließen. Es ist unproblematisch, von links anzufangen und die Produktionen zu raten.*

Ziel	*09*	*09*	*9*	ε
NT-Wort	**S**	**AB**	**B**	
Herleitung	**S**	**AB**	*0***B**	*09*

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. Die erzeugte Herleitung ist eine Linksherleitung.

Bottom-up: *Hierbei muss man die Regeln rückwärts auf den gegebenen String anwenden und versuchen, das Startsymbol der Grammatik zu erreichen. Auch hier ist es sinnvoll, von links her anzufangen:*

09 $\leftarrow$ **A***9* $\leftarrow$ **AB** $\leftarrow$ **S**

Problem auch hier: Man kann evtl. mehrere Regeln anwenden. Man muss außerdem den Teilstring raten, auf den eine Produktion (rückwärts) angewendet werden soll. Man sieht, dass eine Rechtsherleitung gefunden wurde.

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

Beispiel 5.7.3 *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*	
NT-Wort	**S**	**A**	**A**	**A**	
Herleitung	**S**	**A**	*0***A**	*00***A**	*?*

„002“ kann nur aus **B** *hergeleitet werden:*

Ziel	*002*	*002*	*02*	*2*	
NT-Wort	**S**	**B**	**B**	**B**	
Herleitung	**S**	**B**	*0***B**	*00***B**	*002*

*Ein deterministischer Top-Down-Parser müsste bereits beim ersten Zeichen von „002“ wissen, welche Alternative (***A** *oder* **B***) weiterverfolgt werden soll.*

Ein Bottom-Up Parser hat mit diesem Beispiel keine Probleme.

Misslingt eine Herleitung, so muss ein Parser zurücksetzen („Backtracking“).

5.8 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.

5.8.1 Struktur eines rekursiv absteigenden Parsers

1. Der Parser arbeitet Top-Down bzgl. der Grammatik. Das Eingabewort wird von links nach rechts verarbeitet, manchmal wird zurückgesetzt, falls eine Sackgasse erreicht wurde. Der Parser versucht, eine Linksherleitung zu konstruieren.

2. Für jedes Nichtterminal N der Grammatik existiert ein Parser P_N, der die formale Sprache zu N erkennt. Wenn $N \to w_1 \mid \ldots \mid w_n$ die Regeln sind, die zu N gehören, dann arbeitet P_N wie folgt: Alle Worte w_i werden nach und nach (als Alternativen) ausprobiert. Passt keine der Alternativen, wird „Fehlschlag“ zurückgegeben.

   Bei der Prüfung, ob $N \to w_i$ als Regel verwendet werden kann, wird w_i von links nach rechts durchgegangen, und gegebenenfalls rekursiv der Parser zum entsprechenden Nichtterminalen aufgerufen.

3. Der Parser P_σ zum Startsymbol σ ist der Parser zur Grammatik.

4. 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.

Eigenschaften:

- I.a. ist der Zeitaufwand exponentiell.
- Liefert alle Linksherleitungen.

- 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) → ...

- Ist leicht implementierbar. Für eine große Menge von Grammatiken hat man sofort einen Parser.

- Dieses Verfahren kann man auch noch verwenden, wenn die Grammatik kontextfrei ist, aber noch andere Konsistenz-Bedingungen (z.B. Typenbedingungen) zu erfüllen sind, und bei dem diese Bedingungen während der Syntaxanalyse benutzt werden sollen. Man kann es auch verwenden bei nicht kontextfreien Grammatiken.

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

  Effiziente rekursiv-absteigende Parser, z.B. für Programmiersprachen, benötigen i.a.: Erweiterungen wie Vorausschau um einige Zeichen, um die richtige Regel zu bestimmen, und/oder einen Umbau der Grammatik (Optimierung der Grammatik).

5.9 Funktionale Kombinator-Parser

Diese implementieren 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: `String -> (String,R)`

Da rekursiv absteigende Parser auch zurücksetzen sollen, wird einfach eine Liste der möglichen Paare (Rest,Resultat) als Antwort geliefert. Die verzögerte Auswertung sorgt dann für die richtige Reihenfolge der Abarbeitung.
Der Typ ist somit etwas allgemeiner:

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

Im folgenden eine Haskell-Implementierung einer Reihe von vordefinierten Kombinatoren (Funktionen auf Parsern), die es erlauben eine Grammatik direkt hinzuschreiben. Hier sind auch Kombinatoren vorgesehen, die das Resultat von Kombinatoren in einen Syntaxbaum umbauen.

```
module RekParsKomb where
import Data.Char

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 (isSpace x))

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

--  erkennt Zahl
natural :: Integral a => Parser Char a
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 :: Integral a => Parser Char a
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 n-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 5.9.1 *arithmetische Ausdrücke mit folgender Grammatik*

Ex	::=	*Plus*
Plus	::=	*SigZ Plusrest*
PlusRest	::=	$+$ *SigZ PlusRest* $\mid \varepsilon$
SigZ	::=	*B* $\mid -$ *B*
B	::=	*Z* $\mid$ *(Ex)*

```
plusausdruecke = just pAriEx

pAriEx :: Parser Char Integer

pAriEx      = (wsp pAriPlus)
pAriPlus    = (opSeqInf (symbol '+') pAriSigZ) <@ sum
--            direkt implementiert waere das:
--            (pAriSigZ  <*> many ((symbol '+') *> pAriSigZ))  <@ list
pAriZsymbol = naturalex
pAriSigZ    = pAriB  <|> (symbol '-')  *> pAriB <@ (negate)
pAriB       = pAriZsymbol <|> (pack (symbol '(')  pAriEx  (symbol ')'))
```

Ein Parser zur obigen Grammatik, der pro Nichtterminal eine Regel hat und noch das Ergebnis als Zahl sofort berechnet, ist ebenfalls kodiert: Beachte, dass bei `pqplusrest` *eine naive Kodierung ohne sofortige Nachverarbeitung einen Typfehler ergibt.*

```
pqplus ein = let aus = (just pqAriEx) ein
              in if aus == [] then "Fehler in der Eingabe"
                 else show (( snd . head) aus)
pqplusalle = just pqAriEx

pqAriEx :: Parser Char Integer

pqAriEx      = (wsp pqAriPlus)
pqAriPlus    =  pqAriSigZ  <|> ((pqAriSigZ <*> pqplusrest)  <@
              (\(x,f) -> f x))
pqplusrest   = (((((symbol '+')  *> pqAriSigZ) <@ (\x y -> x+y))
              <*> pqplusrest)    <@ (\(f,g) -> f . g))
                  <|> (epsilon <@ (\x -> id))

pqAriZsymbol = naturalex
pqAriSigZ    = pqAriB  <|> (pack (symbol '(') ((symbol '-')
                 *> (pqAriPlus <@ (negate))) (symbol ')'))
pqAriB       = pqAriZsymbol <|> (pack (symbol '(')  pqAriEx  (symbol ')'))
```

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?

5.9.1 Funktionale Kombinator-Parser mit Fehlerbehandlung

Man kann die Kombinatorparser so erweitern, dass eine Fehlerbehandlung möglich ist. Allerdings muss man dazu die Grammatik so abändern, dass man an bestimmten Stellen erkennen kann, dass Fehler eingetreten sind. Zum Beispiel muss in einem arithmetischen Ausdruck immer ein Ausdruck nach einem + folgen.

Als Beispiel nehmen wir die polymorphen Typen, wie sie in Haskell verwendet werden.

Die intuitive, aber linksrekursive Grammatik ist:

AT ::= **AT -> AT** | (**AT**) | [**AT**] | **Var** | **TCA**
TCA ::= **TC** | (**TC AT** ... **AT**) | ($\mathbf{AT}_1$,...,$\mathbf{AT}_n$), $n > 1$

Die umgebaute Grammatik dazu, die nicht mehr linksrekursiv ist, und die auch bereits optimiert ist, sieht so aus:

AT	::=	**NOAR** { **NOARNX** | ε }	
NOARNX	::=	-> **AT**	
NOAR	::=	**Var** | **TCT** | **KLRUND** | **KLECK**	
TCT	::=	**TC NOAR** ... **NOAR**	
KLRUND	::=	(**AT**,...,**AT**)	Mindestens 2-Tupel
KLECK	::=	[**AT**]	

Die Veränderungen sind: (1) die Pfeiltypen sind in einen Anfang und einen Rest zerlegt, der mit `->` startet; (2) Die Klammertypen sind in solche mit runden und mit eckigen Klammern getrennt, so dass man beim Parsen einer Klammer alternativlos das richtige Nichtterminal wählen kann; (3) Die Tupeltypen sind ebenfalls intern nach erster Komponente und die mit einem Komma beginnende Fortsetzung getrennt.

Das Programm dazu unter Benutzung der neuen Kombinatoren `<*>!`, `<*!` und `*>!` , die nach erfolgreichem Parsen des ersten Teils eine Fortsetzung erzwingen und noch einen Fehlertext als Argument haben.

```
parseEQ  = (((parseAT <*>! "= erwartet ") (((parseSymbol '=')
               *>! "zweiter Typ der Gleichung fehlt") parseAT))
              <*! "Ueberfluessige Symbole") end

parseAT = ((parseNOAR <*> (manyex  parseNOARNX))
         <@@ (\(t1,t2) er -> if null t2 then t1 else Fn "->" (t1:t2) er))

parseNOARNX =   ((satisfy tokenIsArrow)
        *>! "Zweiter Funktionstyp nach -> fehlt") parseAT

parseNOAR = parseVar <|> parseTCT <|>  parseKLRUND <|>  parseKLECK

parseNOARTC = parseVar <|> parseTC <|>  parseKLRUND <|>  parseKLECK

parseTCT = (parseTC <*>  (manyex parseNOARTC))
    <@@ (\(t1,t2) er ->
       if null t2 then Fn (getTermTopName t1) [] er
       else Fn (getTermTopName t1) t2 er)

parseKLRUND =
    (parseSymbol '('  *> (parseINKLRUND <*! ") erwartet")  (parseSymbol ')')) <@ id
parseINKLRUND = (parseAT <*> (manyex (((parseSymbol ',')
                       *>! "Typ nach , erwartet") parseAT)))
   <@@ (\(t1,t2) er -> if null t2 then t1
                       else   (Fn ("Tup"++(show ((length t2) +1))) (t1:t2) er))

parseKLECK =  (parseSymbol '[' *> ((parseAT <*! "] erwartet")  (parseSymbol ']'))
        <@@ \t er -> Fn "[]" [t] er)

parseVar = satisfy tokenIsVar <@@  \x er   -> Var (gettoken x) er
parseTC = satisfy tokenIsTC <@@ \x er   -> Fn (gettoken x) [] er

parseSymbol s = satisfy (\t -> gettoken t == [s])
```

5.9.2 Parsen am Beispiel von HTML-Files

Der HTML-Standard bildet die Basis für viele Dokumente die von Web-Browsern angezeigt werden können. Es gibt diesen Standard bereits länger; dieser wurde um 1989 von Tim Berners-Lee am CERN festgelegt und

gilt als ein Start des Internets. Mittlerweile gibt es neuere Versionen des Standards, verschiedene Weiterentwicklungen, wie z.B. XML.

In dieser Vorlesung soll es nur als gutes Beispiel eines strukturierten Textes dienen, den wir mit einem Beispiel-Parser verarbeiten, um die Baumstruktur zu erkennen.

HTML heißt Hyper Text Markup Language. Es ist ein Text, der mit speziell Marken versehen ist, die den Text einerseits strukturieren, so dass man ihn auf eine Webseite anzeigen kann, und andererseits enthält der Text Links, die auf andere solche Dokumente verweisen, und die meist anklickbar sind.

Ein HTML-Dokument hat im wesentlichen drei Abschnitte:

- Dokumenttypdeklaration (Doctype)
  das sind Hinweise an den Browser und geben de Standard an, Z.B. wie in wiki:
  `<!DOCTYPE HTML PUBLIC //W3C//DTD HTML 4.01//ENhttp://www.w3.org/TR/html4/strict.dtd≫`

- HTML-Kopf. Hier steht der Titel der Webseite und evtl. weitere Infos, sogenannte Meta-Informationen, die normalerweise nicht angezeigt werden, Abkürzungen, Layout-Informationen, usw.

  Beispiel für die Syntax ist:

```
    <head>
      <title> Programmierung 2 </title>
      <!-- Evtl. weiteres  -->
  </head>
```

- HTML-Rumpf (body). Enthält den eigentlichen Text und die versteckten Informationen der Webseite.

  Beispielfile, an dem man schon die Syntax ahnen kann:

```
  <body>
    <p>Inhalt der Webseite</p>
    <TABLE BORDER="0" CELLPADDING="0">
       <TR>
        xxxx
          <a href="http://www-stud.informatik.uni-frankfurt.de/~prg2">
              Grundlagen der Programmierung 2 [PRG-2]</a>
       </TR>
     </table>
  </body>
```

Beispiel einer Webseite aus wikipedia:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <title>Titel der Webseite</title>
    <!-- Evtl. weitere Kopfinformationen -->
  </head>
  <body>
    <p>Inhalt der Webseite</p>
  </body>
</html>
```

Die Markierungen im Text sind von der Form `<marke>` gefolgt von Text und dann `</marke>`. Diese wirken wie eine Klammerung und zeigen die Baumstruktur an. In HTML gibt es nur eine feste Anzahl an solchen Markierungen bzw. Tags. Bei diesen kommt es auf die Groß- und Kleinschreibung nicht an. z.B. gibt es `html`, `head`, `body`, `p`, `a`,`table`, `li`, `ul`, usw.

Spezialitäten sind Attributwerte, die teilweise angegeben werden wie in `<a href="http://www-stud.informatik.uni-frankfurt.de/~prg2">` und als Attribut von `<a>` angesehen werden können. Ebenso Kommentare, die mit `<!--` anfangen und mit `-->` beendet werden. Ebenso die Einbettung von Strings.

Ein Beispielparser wird in den Programmen zu PRG2 bereitgestellt, mit dem man html-Seiten parsen kann, wenn diese richtig geklammert sind und

```
parseDokument  =   parseHtmlElementListe

parseHtmlElementListe =  manyex  parseHtmlElement
parseHtmlElement =    (parseHtmlBrackElement
       <!>
     parseHtmlSingleElement
       <!>
     (satisfy (\t -> (((take 2 (gettoken  t)) /= "</"))))
            <@ (\tok -> ST "" [gettoken tok] [])))

parseHtmlBrackElement  =  foldl1   (<!>)
            (map (\tag -> (parseHtmlMarkenElt tag))  bracketTags)
parseHtmlSingleElement =  foldl1   (<!>)
           (map (\tag -> (parseHtmlSinElt tag))  singleTags)

parseHtmlMarkenElt m =
   (((parseMarke m  <*> parseHtmlElementListe) <*! ("Endemarke " ++ m ++ " erwartet"))
        (parseEndMarke m))
         <@ (\(mcontlst,eltlst) -> ST m mcontlst eltlst)

parseHtmlSinElt m = parseMarke m <@ (\mcontlst  -> ST m mcontlst [])


bracketTags = ["HTML","BODY","HEAD","TABLE","P","TD","TR",FONT","TITLE","DFN","LI",
 "DL","DD","DT","UL","LI","B","A","H1","H2","STYLE","DIV","SPAN","FRAMESET","NOFRAMES"]
singleTags = ["!DOCTYPE", "!--", "HR","META", "BASE","BR","FRAME"]
```

Anmerkungen dazu:

- Der File wird als Liste von Elementen geparst,
- Die Elemente sind:
  1. `<tag>  elementliste </tag>` , oder
  2. `<tag>` mit Attributen im Tag, oder
  3. Liste von Strings

Ein Tag kann in den Klammern noch Attribute enthalten, wie z.B. in

```
   <a href="http://www-stud.informatik.uni-frankfurt.de/~prg2">
                   Grundlagen der Programmierung 2 [PRG-2]</a>
```

Es gibt auch Tags, die alleine stehen, ohne Ende-Tag und dafür aber Attribute enthalten.

Beispiel 5.9.2 *Der Parser kann z.B. die Startseite der Vorlesung „ Einführung in die Programmierung 2“parsen.*

Problem in der Praxis ist, das manche Browser auch falsch geklammerte html-Files akzeptieren und auf bestimmte Weise Klammern einfügen. Das Folgeproblem ist, dass dadurch fehlerhafte html-Webseiten in manchen Browsern angezeigt werden, und in manchen nicht, bzw. je nach Klammerergänzung sehen die falschen Seiten verschieden aus.

5.10 Rekursiv-prädiktive Parser

Dies sind optimierte, rekursiv absteigende Parser, für eingeschränkte Grammatiken, mit folgender Eigenschaft:

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

Die Zuordnung des Paares (Lookahead-Symbol, aktuelles Nichtterminal) zu einer Regelalternative muss eindeutig sein, d.h. nimmt man die Menge der rechten Seiten der Regeln zu einem Nichtterminal **A**, dann darf es abhängig vom nächsten Symbol nur genau eine anwendbare Regel geben. Die Anwesenheit von ε-Regeln erfordert eine Sonderbehandlung: diese kommen zum Zug, wenn es für das Lookahead-Symbol keine rechte Seite gibt, und wenn das aktuelle Symbol auf das Nichtterminal folgen kann.

Die Grammatiken, für die die Optimierung angewendet werden kann, nennt man LL(1)-Grammatiken (bzw. -Sprachen): Von **L**inks nach rechts wird die Eingabe abgearbeitet, dabei eine **L**inksherleitung erzeugt wird, und 1 Symbol Vorausschau erlaubt ist. Es ist bekannt, dass man nicht für jede kontextfreie Grammatik bzw. kontextfreie formale Sprache einen rekursiv-prädiktiven Parser konstruieren kann.

Definition 5.10.1 *Sei G eine CFG mit S als Startsymbol.*

- *Sei w ein Wort aus Terminalen und Nichtterminalen.*
  Dann ist $\texttt{first}(w) := \{x \in T \mid w \to^* xv$ *für ein Wort* $v \in T^*\} \cup E_w$ *wobei* $E_w := \{\varepsilon\}$ *wenn* $w \to^* \varepsilon$ *und* $E_w = \emptyset$ *sonst.*
- *Sei* **A** *ein Nichtterminal. Dann ist*
  $\texttt{follow}(\mathbf{A}) := \{x \in T \mid S \to^* v_1 \mathbf{A} x v_2$ *für Worte* $v_1, v_2 \in T^*\}$

Die Mengen `first` und `follow` kann man für alle Nichtterminale leicht berechnen, indem man eine Fixpunktiteration macht. Die Fixpunktiteration berechnet sukzessiv Funktionen $f_i, i = 0, 1, 2, \ldots :\to T \cup \{\varepsilon\}$ auf Nichtterminalen. Diese Funktionen setzt man auf Worte $w \in (T \cup N)^*$ fort wie folgt:

$$\begin{array}{lcl} f_j(aw) & := & \{a\} \text{ für jedes Terminal } a \\ f_j(Aw') & := & f_j(A) \text{ wenn } \varepsilon \notin f_j(A) \\ f_j(Aw') & := & (f_j(A) \setminus \{\varepsilon\}) \cup f_j(w') \text{ wenn } \varepsilon \in f_j(A) \\ f_j(\varepsilon) & := & \{\varepsilon\} \end{array}$$

Die Fixpunktiteration für `first` läuft wie folgt ab:

$$\begin{array}{lcl} f_0(A_i) & := & \emptyset \quad \text{für alle } i \\ f_{j+1}(A_i) & := & \bigcup_i f_j(w_i) \text{ wenn } A_i ::= w_1 \mid \ldots \mid w_k \text{ die Regeln zu } A \text{ sind.} \end{array}$$

Die Iteration stoppt, wenn $f_{j+1}(A) = f_j(A)$ für alle Nichtterminale A. Das letzte f_j ist dann die Funktion `first`.
Die Fixpunktiteration zur Berechnung der `follow`-Mengen ist wie folgt:

$g_0(A_i) := \emptyset$ für alle i

$g_{j+1}(A_i) := \bigcup B_{R,h}$, mit folgenden Mengen für jede Regel $R : A_k \to wA_iw'$:

$$B_{R,1} := \begin{cases} \texttt{first}(w') \setminus \{\varepsilon\}, & \text{wenn } w' \neq \varepsilon \\ \emptyset, & \text{wenn } w' = \varepsilon \end{cases}$$

$$B_{R,2} := \begin{cases} g_j(A_k) & \text{wenn } \varepsilon \in \texttt{first}(w') \\ \emptyset, & \text{wenn } \varepsilon \notin \texttt{first}(w') \end{cases}$$

Die Iteration stoppt, wenn $g_{j+1}(A) = g_j(A)$ für alle Nichtterminale A. Das letzte g_j ist dann die Funktion `follow`.

Beachte, dass $B_{R,2} = g_j(A_k)$ für jede Regel R mit $A_k \to wA_i$.

Ein alternative Definition für $g_{j+1}(A_i)$ ist:

$g_{j+1}(A_i) :=$ Vereinigung folgender Mengen:

$\texttt{first}(w') \cap T$	für jede Regel $A_k \to wA_iw'$
$g_j(A_k)$	für jede Regel $A_k \to wA_iw'$ mit $\varepsilon \in \texttt{first}(w')$

Beispiel 5.10.2 *Betrachte folgende Grammatik. Hierzu geben wir die Berechnung der first- und follow-Mengen an:*

Ex	*::=*	**Plus**
Plus	*::=*	**SigZ Plusrest**
PlusRest	*::=*	*+* **SigZ PlusRest** | *eps*
SigZ	*::=*	**B** | *-* **B**
B	*::=*	**Z** | *(* **Ex** *)*
Z	*::=*	*0* | *...* | *9*

Zuerst die Berechnung der first-Mengen zu allen Nichtterminalen.

Ex	**Plus**	**Plus Rest**	**SigZ**	**B**	**Z**
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\emptyset$	$\emptyset$	+, ε	-	(	0,...,9
$\emptyset$	-	+, ε	-, (	0,...,9, (	0,...,9
-	-, (	+, ε	0,...,9, (, -	0,...,9, (	0,...,9
-, (	0,...,9, (, -	+, ε	0,...,9, (, -	0,...,9, (	0,...,9
0,...,9, (, -	0,...,9, (, -	+, ε	0,...,9, (, -	0,...,9, (	0,...,9

Und hier die Berechnung der follow-Mengen zu allen Nichtterminalen auf der Basis der eben berechneten first-Mengen.

Ex	**Plus**	**Plus Rest**	**SigZ**	**B**	**Z**
)	$\emptyset$	$\emptyset$	+	$\emptyset$	$\emptyset$
)	)	$\emptyset$	+	+	$\emptyset$
)	)	)	+	+	+
)	)	)	+,)	+	+
)	)	)	+,)	+,)	+
)	)	)	+,)	+,)	+,)

Die first- und follow-Mengen kann man zur Vermeidung von Backtracking verwenden: insbesondere zur Konstruktion eines rekursiv-prädiktiven Parsers:

1. Gibt es für **A** die Regeln $\mathbf{A} ::= w_1 \mid \ldots \mid w_n$ und sind alle Mengen $\texttt{first}(w_i)$ paarweise disjunkt, so kann man anhand des ersten Symbols a des zu parsenden Wortes die richtige Produktion $A ::= w_i$ auswählen, falls $a \in \texttt{first}(w_i)$.

2. Gibt es keine passende $\texttt{first}(w_i)$ -Menge einer rechten Seite einer Regel für A, aber maximal eine Regel $A ::= w$, so dass $\varepsilon \in \texttt{first}(w)$, und das Lookahead-Symbol ist in $\texttt{follow}(A)$, dann nimmt man diese Regel.

3. Wenn auch dann keine passende Alternative existiert, wird mit Fehler abgebrochen.

Wenn diese Auswahl immer eindeutig getroffen werden kann, dann ist die Grammatik eine LL(1)-Grammatik, und der rekursive Parser ist deterministisch. Diese Eigenschaft kann man direkt aus der Grammatik berechnen, indem man für jedes Nichtterminal A folgendes testet:
Wenn $\varepsilon \notin \texttt{first}(A)$, dann müssen die `first`-Mengen aller rechten Seiten von Regeln zu A paarweise disjunkt sein. Wenn $\varepsilon \in \texttt{first}(A)$, dann müssen die `first`-Mengen der rechten Seiten von Regeln zu A und $\texttt{follow}(A)$ paarweise disjunkt sein. Höchstens eine rechte Seite einer Regel zu `A` darf eine `first`-Menge mit ε haben.

Damit ist auch eine frühe Fehlererkennung für LL(1)-Grammatiken möglich: man kann abbrechen, wenn es zu einem Nichtterminal, das gerade angewendet werden soll, kein passendes Symbol in den `first` und `follow`-Mengen unter den angegebenen Bedingungen gibt.

Die `first` und `follow`-Mengen sind auch dann sinnvoll einsetzbar, wenn die Grammatik nicht LL(1) ist, da man dann ebenfalls viele nutzlose Verzweigungen verhindern kann.

Beispiel 5.10.3 *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** $\mid -$ **Term Rest** $\mid \varepsilon$
Term	$::=$	$0 \mid \ldots \mid 9$

Berechne die `first`*-Mengen:*

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

Berechne die `follow`*-Mengen:*

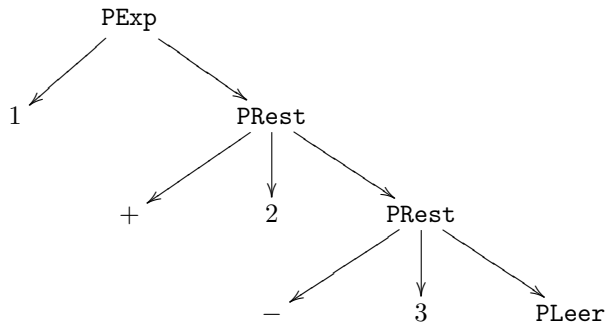
- $\texttt{follow}($**Expr**$) = \emptyset$.
- $\texttt{follow}($**Rest**$) = \emptyset$.
- $\texttt{follow}($**Term**$) = \{+,-\}$.

Diese Grammatik hat die LL(1)-Eigenschaft: u.a. ist $\texttt{follow}($**Rest**$) = \emptyset$*. Also kann man einen deterministischen und rekursiv-prädiktiven Parser angeben.*

Beispiel 5.10.4 *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; rest =  nil"
parseTerm eingabe@(symbol:rest) =
         if not  (symbol `elem` ['0'..'9'] )
         then  error ("Ziffer erwartet; rest = " ++ eingabe)
         else (rest,Pblatt (fst (head (reads [symbol]))))

parseRest [] = ([],PLeer)
parseRest eingabe@(operator: rest) =
   if not  (operator `elem` ['+','-'] )
         then  error ("+ oder - erwartet; rest = " ++ eingabe)
   else
     let (rest1,resultat1) = parseTerm rest
         (rest2,resultat2) = parseRest rest1
       in (rest2, PRest operator resultat1 resultat2)

parseExp :: [Char] -> (String,Pbaum)
parseExp [] = error ("Ziffer erwartet; rest = nil")
parseExp (eingabe@(symbol: rest)) =
   if not  (symbol `elem` ['0'..'9'] )
         then  error ("Ziffer erwartet; rest = " ++ eingabe)
   else
     let (rest1,resultat1) = parseTerm eingabe
         (rest2,resultat2) = parseRest rest1
       in  (rest2, PExp  resultat1 resultat2)


test1::(String,Pbaum)
test1 = parseExp "1 + 2 - 3"         --   blanks sind noch da!!
test2::(String,Pbaum)
test2 = parseExp "1+2-3"

Main> test2
("",PExp (Pblatt 1) (PRest '+' (Pblatt 2)
  (PRest '-' (Pblatt 3) PLeer))) :: (String,Pbaum)
```

Der von diesem Parser aufgebaute Baum sieht so aus. Er entspricht offenbar noch nicht der gewünschten Struktur des arithmetischen Ausdrucks. Um das zu erreichen, braucht man noch eine Nachbearbeitung.



5.10.1 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 für nicht-LL(1)-Sprachen 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.

5.11 Operationen auf Grammatiken

(dieses Kapitel ist nicht Teil des Prüfungsstoffes 2017, aber das nächste)

Es gibt verschiedene Methoden zur Abänderung der Regelmenge einer Grammatik, die die zugehörigen formalen Sprache erhalten. Diese haben verschiedene Zielrichtungen: Normalisierungen, Vereinfachungen, Elimination bestimmter Konstrukte und können verwendet werden, um das Erzeugen eines Parsers zu erleichtern. Unter anderem kann manchmal eine Grammatik so umkonstruiert werden, dass sie in einem rekursiv-prädiktiven Parser benutzt werden kann. Diese Transformationen verändern normalerweise die zugehörigen Parsebäume, so dass man die operationale Semantik, die ja am Parsebaum festgemacht ist, ebenfalls mitändern muss. Das ist teilweise offensichtlich, muss aber in der Regel nach der Transformation von Hand spezifiziert werden. Zu beachten ist auch, dass die Transformationen die Grammatik vergrößern können, so dass man im schlimmsten Fall zwar einen Parser erhält, der lineare Laufzeit bzgl. der Eingabe hat, aber vorher die Grammatik exponentiell vergrößert hat, so dass der Gewinn möglicherweise erst bei sehr großen Strings sichtbar wird, oder dass der Parser selbst so groß ist, dass er sich nicht mehr kompilieren lässt.

5.11.1 Normalisierungen

Ein kontextfreie Grammatik ist in **Chomsky-Normalform**, wenn alle Produktionen die Form $A ::= a$ oder $A ::= BC$ haben, wobei wobei A, B und C Nichtterminale sind und a ein Terminalsymbol ist.

Ein kontextfreie Grammatik ist in Greibach-Normalform, wenn alle Produktionen die Form $A ::= aB_1 \dots B_k$ haben, wobei wobei A, B_i Nichtterminale sind und a ein Terminalsymbol ist.

5.11.2 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:

1. Feststellen, welche Nichtterminale ein ε in ihrer erzeugten Sprache haben (die löschbar sind), und für welche Nichtterminale die erzeugte formale Sprache Teilmenge von $\{\varepsilon\}$ ist (die leeren Nichtterminale). Dies macht man mit einer Fixpunktiteration.

2. Erzeuge folgende neue Regelmenge: Wenn $A \to X_1 \dots X_n$ eine Regel ist, dann sind in der neuen Grammatik alle Regeln $A \to \alpha_1 \dots \alpha_n$, 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 bei dieser Transformation exponentiell wachsen.

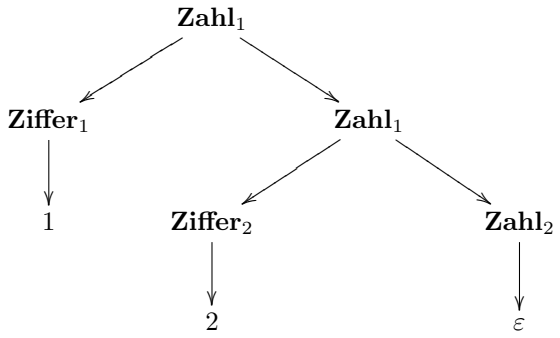
Diese Operation verändert die Menge der Regeln, somit die Herleitungsbäume und kann damit eine an den Herleitungsbäumen festgemachte Semantik verändern. Oft ist die Semantik allerdings so definiert, dass die ε-Resultate keine Rolle spielen, d.h. dass diese wie ein neutrales Element im Herleitungsbaum 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 5.11.1 *Eine Grammatik G_1, die positive ganze Zahlen beschreibt ist:*
Zahl ::= **Ziffer Zahl** | ε
Ziffer ::= 1 | ... | 9 | 0

Zahl *ist ein löschbares Nichtterminal.*

Wenn wir diese ε-frei machen, erhalten wir G_2:
Zahl ::= **Ziffer Zahl** | **Ziffer**
Ziffer ::= 1 | ... | 9 | 0

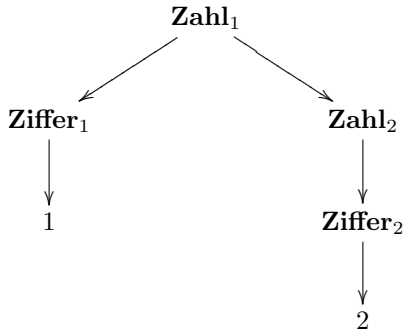
Beispiel 5.11.2 *Der Effekt der Epsilon-Elimination: auf Parsebäume hier beispielhaft demonstriert, wobei wir die Grammatiken von oben nehmen, um das Startsymbol* **S** *erweitert.*
S ::= **Zahl**
Zahl ::= **Ziffer Zahl** | ε
Ziffer ::= 1 | ... | 9 | 0
G_1-Parsebaum für das Wort „12“:



Die Semantik für G_1-Parsebäume kann man so definieren:
Zahl$_1$ *ist vom Typ:* `Int->(Int,Int)->(Int,Int)`.
Der Wert wird intern durch Paare dargestellt: Die erste Komponente des Paares ist die 10-er Potenz die der Stelligkeit entspricht, die zweite Komponente ist der Zahlenwert des rechten Baumes. Wir erlauben der Einfachheit halber in den Lambda-Ausdrücken Pattern anstelle der Argumente.

S	$\mapsto$	$\lambda(x,y)\ .\ y$	
Zahl$_1$	$\mapsto$	$\lambda x,(y1,y2)\ .\ (10*y1, y1*x+y2)$	
Zahl$_2$	$\mapsto$	$\lambda x\ .\ (1,0)$	
Ziffer$_n$	$\mapsto$	$\lambda x\ .\ n$	(*Alternativ:* $\lambda x\ .\ \text{wert}(x)$)
		Haskell: `\x ->(read [x])::Int`	

Der Parsebaum zum Wort „12“ bzgl G_2 sieht so aus:



Eine Semantik für G_2-Parsebäume kann man so definieren:

S	$\mapsto$	$\lambda(x,y)\ .\ y$	
Zahl$_1$	$\mapsto$	$\lambda x,(y1,y2)\ .\ (10*y1, y1*x+y2)$	
Zahl$_2$	$\mapsto$	$\lambda x\ .\ (10,x)$	
Ziffer$_n$	$\mapsto$	$\lambda x\ .\ n$	(*Alternativ:* $\lambda x\ .\ \text{wert}(x)$)

Die Epsilon-Elimination bewirken, dass die Zahlberechnung verändert werden muss. Es entspricht einer Verkürzung der Liste um 1 Element, so dass Weitergabe der Zehnerpotenz mit 10 startet statt mit 1 und bei der Berechnung ε dem neutralem Element, der 0, entspricht.

5.11.3 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 5.11.3 *Direkte Links-Rekursion* $\mathbf{A} ::= \mathbf{A}w$

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

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

Definition 5.11.4 *Eine Grammatik G ist* linksrekursiv*, wenn es eine Folge von Nichtterminalen* $A_1, \ldots, A_n$ *gibt und eine Folge von Regeln der Grammatik der Form* $A_1 ::= A_2 w_1, A_2 ::= A_3 w_2, \ldots, A_n ::= A_1 w_n$.

Entsprechend kann man rechtsrekursiv definieren

Diese Linksrekursion ist durchaus sinnvoll bei der Spezifikation von formalen Sprachen, sie ist nur störend bei der Methode des rekursiven Abstiegs.
Eine Methode zur Abhilfe ist das Einführen neuer Nichtterminale. Wir betrachten den Fall, dass es für **A** genau die folgende Regeln gibt:

$\mathbf{A} ::= \mathbf{A}\alpha \mid \beta$

Hierbei soll β nicht mit **A** beginnen, aber α, β können noch **A** enthalten. Zusätzlich soll $\alpha\beta \neq \varepsilon$ gelten.

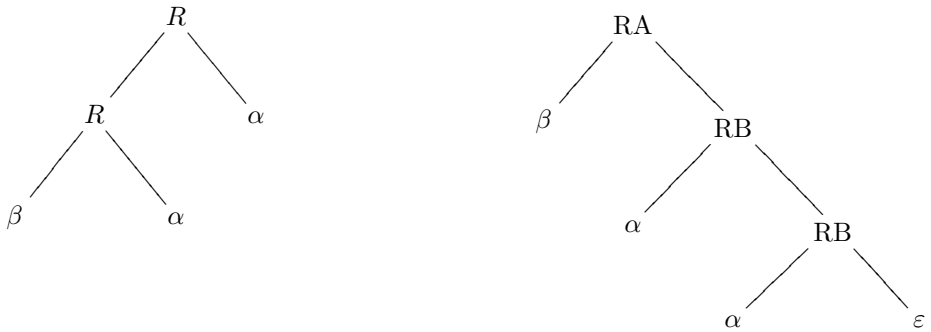
Die Transformation ist folgendermaßen:
Man fügt das neue Nichtterminal **B** hinzu. Die obigen zwei Regeln werden ersetzt durch:

A ::= $\beta \mid \beta\mathbf{B}$
B ::= $\alpha\ \mathbf{B} \mid \alpha$

Jetzt ist weder **A** noch **B** linksrekursiv. Man kann die Fälle, dass es mehr Regeln für **A** gibt, leicht auf auf den obigen Fall zurückführen, bzw. als durch eine leichte Erweiterung der Vorgehensweise behandeln.
Wir geben eine Begründung für die Korrektheit dieser Umwandlung im Sinne der Erhaltung der formalen Sprache. Dazu nehmen wir an, dass β und α nicht mit **A** starten.

Vor der Transformation entspricht die zu **A** gehörige formale Sprache gerade $\beta\alpha^*$.

Nach der Transformation gilt:
Die zu **B** gehörige Sprache entspricht α^+. Die zu **A** gehörige Sprache entspricht $\beta\alpha^*$. D.h. die jeweils zu **A** gehörige formale Sprache hat sich nicht verändert.
Wir betrachten die Syntaxbäume vorher und nachher für ein Beispiel:



Eine Möglichkeit, die operationale Semantik bzgl des neuen Parsebaumes zu definieren, ist es, eine Abbildung der neuen auf die alten Parsebäume anzugeben.

Eine Betrachtung der zugehörigen semantischen Abbildung in diesem Fall, wie auch in anderen zeigt, dass man durch Beachten der Assoziativität des passenden Operators eine direkte Abbildung auf den neuen Parsebäumen angeben kann. 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. Dies entspricht den Vorstellungen bei Operatoren wie $+, *$, append und concat. Zu beachten ist, dass z.B. die Multiplikation von Gleitkommazahlen mit Rundung nicht assoziativ ist, und dass diese Umstellung mittels Assoziativität dann die operationale Semantik verändert.

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 5.11.5 *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 einen anderen Wert. Schaut man genauer hin, könnte man Abhilfe schaffen durch die Interpretation:* $1 + (-2) + (-3)$*, die einen assoziativen Operator verwendet.*

Beispiel 5.11.6 *Für eine indirekte Links-Rekursion:*

A ::= **S**a | b
S ::= **B**c | **A**d | e
B ::= $f \ldots$

Hier wird ein rekursiver Parser bei manchen (nicht allen) Worten, die zu $L(G)$ *gehören, in eine Schleife geraten. Z.B. „fc“ 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 $\mathbf{A} \to^* \mathbf{A}\alpha$ gibt.

Jede Linksrekursivität kann durch Transformation der Grammatik beseitigt werden, unter Beibehaltung der erzeugten formalen Sprache. Diese Transformationen bewirken i.a. eine Veränderung der Herleitungsbäume, so dass man die semantische Abbildung entsprechend anpassen muss.

5.11.4 Beseitigung von Mehrdeutigkeiten

Klassisches Beispiel ist das „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 Mehrdeutigkeiten kann man oft 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**

5.11.5 Links-Faktorisierung

Ein weiteres Problem, das einen effizienten Einsatz rekursiv absteigender Parser behindert, sind die gleichen Anfänge verschiedener Regelalternativen: z.B.lässt sich mittels `first()` nicht immer entscheiden, welche Produktion anzuwenden ist. Die gleichen Anfänge bewirken teilweise ein zu weites Zurücksetzen. Der Gesamteffekt ist ein exponentielles Ansteigen der verschiedenen Suchmöglichkeiten.

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 5.11.7

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.

Sinnvoll ist es auch gleiche Anfänge rechter Seiten von Regeln für verschiedene Nichtterminale zusammenfassen. Damit diese Optimierung etwas bringt, muss man das Ausklammern auch über mehrere Regeln durchziehen:

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

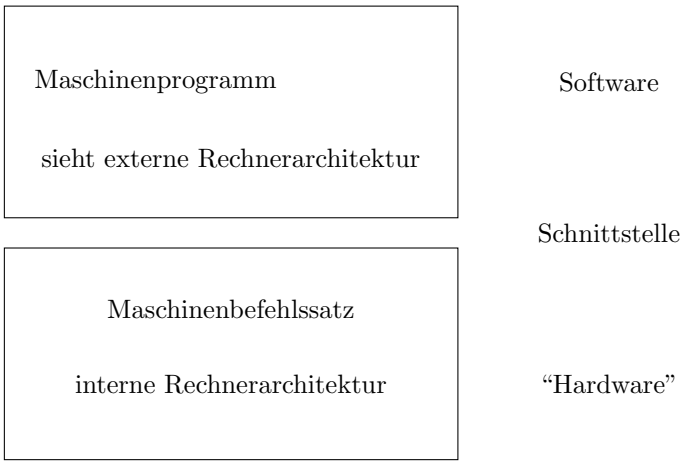
wird zu:

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

Es ist nicht so schwer zu erkennen, dass die formalen Sprachen die gleichen sind. Die Nichtterminale haben sich etwa verdoppelt.

5.12 Kodegenerierung, Rechnerarchitekturen, abstrakte Maschinen

(Ist Prüfungsstoff)



- Die *externe Architektur* legt durch den Maschinenbefehlssatz fest, wie ein Rechner von der Seite der Software her angesprochen werden kann. Dies ist die für den Kodegenerator maßgebliche Schnittstelle.
- Die *interne Architektur* bezieht sich auf den internen Aufbau und das Organisationsprinzip eines Rechners.
  (Dies ist oft keine Hardware mehr sondern auch einen mittels Software nachgebildete Hardwareschnittstelle.)

Typische externe Architekturen:

Stackmaschinen: Maschinenbefehle beziehen sich implizit auf einen Stack. Das Ergebnis von Operationen wird im Stack abgelegt. Dieser wird z.T. durch Register in der CPU nachgebildet.

Akkumulatormaschinen: impliziter Operand ist ein Akkumulator-Register. I.a. gibt es ein weiteren expliziten Operand. Ziel ist ebenfalls der Akkumulator.

Registersatzmaschinen: typisch sind zwei oder drei Operanden (Zwei- bzw. Dreiadressbefehle), die Register oder Hauptspeicheradressen sein können. (RISC-Befehlssätze, s.u.)

Beispiel 5.12.1 $a := b + c$

Stack	*Akkumulator*	*Registersatz*
PUSH b	*LOAD c*	*LOAD R1,c*
PUSH c	*ADD b*	*ADD R1,b*
ADD	*STORE a*	*STORE a,R1*

Abarbeitung:

1. **Stackmaschine**

Befehl	*Stack*
...	...,
`PUSH b`	..., *b*
`PUSH c`	..., *b*, *c*
`ADD`	..., *b*+*c*

2. **Akkumulatormaschine:**

Befehl	*Zustand*
...	*Akku = ?*
`LOAD c`	*Akku = c*
`ADD b`	*Akku = c+b*
`STORE a`	*Akku = c+b, (Adresse a wird überschrieben)*

3. **Registersatzmaschine**

Befehl	*Zustand*	*Anmerkung*
...	*R1 = ?*	
`LOAD R1,c`	*R1 = c, R2 = ?,*	
`ADD R1,b`	*R1 = c', R2 = ?, ...*	*wobei* $c' := c + b$
`STORE a,R1`	*R1 = c', R2 = ?, ...*	*Speicheradresse a wird mit dem Inhalt von R1 überschrieben*

5.13 Kode-Erzeugung und abstrakte Maschinen

Die Kodeerzeugung geschieht im allgemeinen in zwei Schritten: Man wählt eine Architektur aus, die bereits rechnernah ist, konzipiert dann eine passende abstrakte Maschine und modelliert danach den Zwischencode. Dies entspricht dem Entwurf eines Programms für die abstrakte Maschine.

Zu einer **abstrakten Maschine** gehört:

Modellierung des globalen Speichers: Dies ist i.a. eine Kombination von Hauptspeicher, Stack, Register, Akkumulator, usw.

abstrakte Maschinensprache Damit kann man Programme schreiben. Die Möglichkeiten variieren. Normalerweise ein imperatives und prozedurales Programm mit Zugriff auf die verwendeten Speicherobjekte.

operationale Semantik D.h ein Interpreter für die abstrakte Maschinenbefehle ist spezifiziert.

Abstrakte Maschinen unterscheiden sich von höheren Programmiersprachen dadurch, dass die Operationen explizit(er) beschrieben werden. Zum Beispiel wird beim Anwenden einer Funktion auf Argumente genauer gesagt wo die Argumente zu finden sind und wie zum Beispiel die formalen Parameter des Rumpfes durch die Argumente ersetzt werden.

5.13.1 Stackmaschine als abstrakte Maschine: Zwischencode-Erzeugung

Datenstrukturen, die den Zustand der abstrakten Maschine ausmachen:

- Stack (übersetzt als Keller)
- Programm,
- Programmzeiger
- Hauptspeicher

Der Stack kann nur von oben her angesprochen werden. D.h. er wird von oben her adressiert.

Die einfachsten Stackbefehle sind die Operationen: `push, pop`, die ein Element auf dem Stack ablegen bzw. eins entfernen. Diese Datenstruktur ist sehr ähnlich zu einer Liste. Allerdings werden später auch andere Operationen dazu kommen. Z.B. Lesen des i-ten (von oben) Stackeintrags, Additionen auf den obersten Stackelementen, Löschen mehrerer Stackeinträge, usw.

Um mit einer einfachen Stackmaschine operieren zu können, nehmen wir zunächst an, dass auf dem Stack nur ganze Zahlen stehen dürfen:

Operationen, die im Stackprogramm möglich sind:

`pushK` n	n ist Zahlenkonstante, die auf den Stack obendrauf gelegt wird.
`pop`	das oberste Element des Stacks wird entfernt.
`push` i	i ist eine Zahl als Stackadresse; 0 = oberstes, 1 = zweites, usw. diese Element des Stacks wird oben auf den Stack kopiert.
$+, -, *, /$	arithmetische Operationen: Jeweils die obersten beiden Elemente des Stacks werden verknüpft und danach vom Stack entfernt. D.h. aus $\ldots; a; b$ wird unter der Operation $-$: der Stack $\ldots, a - b$.
`print`	Gebe das Ergebnis aus: drucke das oberste Element des Stacks und lösche das gedruckte Element.

Hier ist es **wichtig** auf die Beschränkungen zu achten: Z.B., dass i, k konstant sind in den Befehlen [pushK k] [push i]. Es ist unproblematisch, eine entsprechende abstrakte Maschine für variable Werte zu entwerfen. Allerdings entfernt sich diese dann von einer echten Rechnerarchitektur, da dies Referenzen im Programmcode erfordert.

Beispiel 5.13.1 *Damit kann man bereits ein Programm zur Berechnung von arithmetischen Ausdrücken schreiben: Berechne* $1 + 3 - 5 * 6$*:*

```
pushK 1
pushK 3
+
pushK 5
pushK 6
*
-
print
```

Die operationale Semantik dieser Stackbefehle sieht formalisiert so aus: Es gibt einen **Interpreter** I mit zwei Argumenten:

1. das aktuelle Programm als Liste. Der nächste (aktuelle) Befehl ist das erste Element der Liste.
2. Den Stack als Liste.

Die **operationale Semantik** $I(.,.)$ der jetzigen (rudimentären) Stackmaschine sieht in Haskell-Notation dann so aus:

	Programm	*Stack*		
I	(`pop` : *programmr*)	(a : *stackr*)	$\longrightarrow$	I *programmr stackr*
I	(`pushK` k : *programmr*)	`stack`	$\longrightarrow$	I *programmr* (k : *stack*)
I	(`push` i : *programmr*)	*stack*	$\longrightarrow$	I *programmr* (*stack* !! i :: *stack*
I	($+$: *programmr*)	(a : b : *stackr*)	$\longrightarrow$	I *programmr* ($b + a$: *stackr*)
I	($-$: *programmr*)	(a : b : *stackr*)	$\longrightarrow$	I *programmr* ($b - a$: *stackr*)
I	(`print` : *rest*)	(a : *stackr*)	$\longrightarrow$	I *programmr* (*stackr*) `drucke` a als Seiteneffekt.

Es ist mit einer Argument- und Resultatsübergabekonvention auch möglich, Funktionsaufrufe durchzuführen: Meist ist die Vereinbarung so, dass die Argumente einer Funktion bei deren Aufruf sich auf dem Stack befinden, wobei das letzte Argument oben ist. Das Resultat wird nach Beendigung der Auswertung der Funktion oben auf dem Stack abgelegt, wobei die Argumente vorher konsumiert (d.h.vom Stack gelöscht) wurden.

Beispiel 5.13.2 *Berechne $x^2 + y^2$, wenn der Stack am Anfang $\ldots, x, y$ ist:*

aktueller Befehl	*Stackinhalt*	
	$r; x; y$	
`push` *0*	$r; x; y; y$	
$*$	$r; x; y_1$	*wobei* $y_1 = y * y$
`push` *1*	$r; x; y_1; x$	
`push` *0*	$r; x; y_1; x; x$	
$*$	$r; x; y_1; x_1$	*wobei* $x_1 = x * x$
$+$	$r; x; z$	*wobei* $z = y_1 + x_1$
`print`	$r; x$	

Diese Implementierung der Funktion hält nicht ganz die Resultatsübergabekonvention ein, da auf dem Stack das Argument `x` *noch vorhanden ist. Außerdem wird $y^2 + x^2$ berechnet statt $x^2 + y^2$, was aber bei ganzen Zahlen unproblematisch ist.*

Um Übergabekonventionen einzuhalten, erlauben wir in der Stackmaschine als **zusätzlichen Befehl:**

`slide` m n Schiebe die obersten m Elemente um n nach unten. D.h. n Elemente unter den m oberen verschwinden.

Operationale Semantik des `slide`-Befehls:

$$I\ (\texttt{slide}\ m\ n : programmr)\ stack \longrightarrow I\ programmr\ (take\ m\ stack +\!\!+ (drop\ (n+m)\ stack))$$

Beispiel 5.13.3 *Berechne $x^2 + y^2$, wenn der Stack am Anfang $\ldots, x, y$ ist. Diesmal mit richtiger Übergabekonvention des Resultats.*

aktueller Befehl	*Stackinhalt*	
	$r; x; y$	
`push` *0*	$r; x; y; y$	
$*$	$r; x; y_1$	*wobei* $y_1 = y * y$
`push` *1*	$r; x; y_1; x$	
`push` *0*	$r; x; y_1; x; x$	
$*$	$r; x; y_1; x_1$	*wobei* $x_1 = x * x$
$+$	$r; x; z$	*wobei* $z = y_1 + x_1$
`slide` *1 1*	$r; z$	
`push` *0*	$r; z; z$	
`print`	$r; z$	

Um auch Fallunterscheidungen und Sprünge zu programmieren, erlauben wir in der Stackmaschine als **zusätzliche Befehle und Markierungen:**

`marke`. symbolische Sprungmarke im Programm.

`branchz marke` Wenn auf dem Stack eine Null steht, springe zu marke und lösche das oberste Element

`jump marke` Springe zu `marke`.

Mit den Marken wurde jetzt eine symbolische Referenz wieder in den abstrakten Maschinenkode eingefügt. Man kann jetzt verschiedene Sichtweisen vertreten: a) Die Symbole stehen für Adressen und werden nur für den Programmierer angezeigt.

b) Die Symbole sind tatsächlich vorhanden. In diesem Fall müssen in einem anschließenden Kodeerzeugungsschritt diese symbolischen Referenzen wieder in eine explizite Adressierung umgewandelt werden. Dies geschieht einfach durch Angabe der Indizes in der Liste des Programms. Hier muss man darauf achten, dass man diese symbolischen Marken komplett aus einem Programm entfernen kann, und die Indizes als Ziele der Sprünge dem Programm nach Löschung angepasst sein müssen.

Die operationale Semantik muss dazu erweitert werden: $I(.,.,.)$ ist jetzt dreistellig; das dritte Element ist

jeweils eine Kopie des ganzen Programms.

	Prog	*Stack*	*ProgKopie*		
I	(*marke* : *prog*)	*stack*	*progs*	$\longrightarrow$	*I prog stackprogs*
I	((`branchz` *marke*) : *prog*)	(*top* : *stack*)	*progs*	$\longrightarrow$	
		`if` (0 == *top*) `then` *I* (`dropWhile` (*marke*/ =) *progs*) *stack progs*			
		`else` *I prog stack progs*			
I	((`jump` *marke*) : *prog*)	*stack*	*progs*	$\longrightarrow$	*I* (`dropWhile` (*marke*/ =) *progs*) *stack progs*

Beispiel 5.13.4 *Mit den obigen Stackmaschinenbefehlen kann man die Fakultätsfunktion definieren, siehe Programm in Abb. 5.1. Am Anfang steht das Argument auf dem Stack. Danach wird in der Schleife das Ergebnis oben auf dem Stack abgelegt, der Zähler ist tiefer im Stack.*

```
anfang.
push 0
loop.
push 1
pushK 1
-
branchz ende
push 1
pushK 1
-
push 1
push 1
*
slide 2 2
jump loop
ende.
slide 1 1
print
```

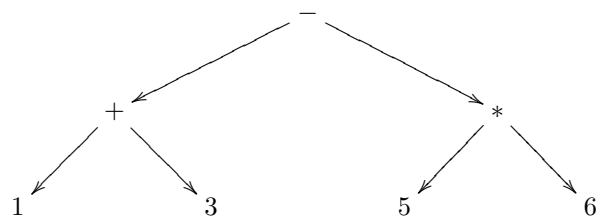
Abbildung 5.1: Stackmaschinenprogramm zur Fakultät.

Das ergibt folgenden Programmablauf, wenn der Stack am Anfang die 3 enthält:

Stack	*Befehl*	*Stack*	*Befehl*	*Stack*	*Befehl*
3	`push` *0*	*2,6*	`jump` *loop*	*1,6*	`jump` *loop*
3,3	`push` *1*	*2,6*	`push` *1*	*1,6*	`push` *1*
3,3,3	`pushK` *1*	*2,6,2*	`pushK` *1*	*1,6,1*	`pushK` *1*
3,3,3,1	-	*2,6,2,1*	-	*1,6,1,1*	-
3,3,2	`branchz` *e*	*2,6,1*	`branchz` *e*	*1,6,0*	`branchz` *e*
3,3	`push` *1*	*2,6*	`push` *1*	*1,6*	`slide` *1 1*
3,3,3	`pushK` *1*	*2,6,2*	`pushK` *1*	*6*	
3,3,3,1	-	*2,6,2,1*	-		
3,3,2	`push` *1*	*2,6,1*	`push` *1*		
3,3,2,3	`push` *1*	*2,6,1,6*	`push` *1*		
3,3,2,3,2	*	*2,6,1,6,1*	*		
3,3,2,6	`slide` *2 2*	*2,6,1,6*	`slide` *2 2*		

Beispiel 5.13.5 *Wie kann man Kode für eine Stackmaschine erzeugen?*
Wir betrachten den Syntaxbaum für den Ausdruck $1 + 3 - 5 * 6$.
Der Syntaxbaum dazu in Präfixschreibweise ist:

minus (plus 1 3) (mal 5 6)



Daraus kann man mittels eines rekursiven Programms das Stackprogramm erzeugen:
minus s t → *Code für s; Code für t;* −
mal s t → *Code für s; Code für t;* ∗
plus s t → *Code für s; Code für t;* +
Konstante → `pushK` *Konstante*
Dies ergibt: Code(plus 1 3); Code(mal 5 6); −
Das Stackmaschinenprogramm ist dann:
`pushK` *1;* `pushK` *3;* +*;* `pushK` *5;* `pushK` *6;* ∗*;* −

5.13.2 Speicheradressierung

Wir erweitern jetzt die Möglichkeiten der Stackmaschine, indem wir zusätzlich einen globalen Speicher (heap, Halde) erlauben, der Zahlen aufnehmen kann, pro Speicherplatz eine Zahl. Die Notation ist als Array $SP[0..]$. Diese Zahlen sind adressierbar mittels einer Adresse (Index) die ebenfalls eine Zahl ist. Der Einfachheit halber erlauben wir, dass Zahlen und Indizes nicht unterscheidbar sind auf dem Stack.
Damit besteht der innere Zustand der Stackmaschine jetzt aus:

- Stack
- Programm,
- Programmzeiger
- globaler Speicher $SP[.]$

Es gibt allerdings die Alternativen, diese Adressen erkennbar zu machen, entweder durch einen eigenen Stack oder durch eine extra Markierung als Adresse. Dieser Extraaufwand für die Markierung hat durchaus praktischen Nutzen. Hierdurch kann ein garbage collector bei jedem Zustand der Stackmaschine noch die referenzierten von unreferenzierten Adressen unterscheiden.

Folgende **Extra Befehle** bewirken eine indirekte Adressierung des Hauptspeichers:

`iget` Der Wert $SP[Stack[0]]$ wird auf den obersten Platz des Stacks abgelegt. Die Adresse auf dem Stack gelöscht.

`iput` Der Wert $Stack[0]$ wird nach $SP[Stack[1]]$ kopiert. Die obersten zwei Elemente des Stacks werden gelöscht.

Ein Interpreter I braucht jetzt ein weiteres Argument: Den Hauptspeicher als Array. Also hat $I(.,.,.,.)$ jetzt 4 Argument: Programm, Stack, Speicherarray, Programmkopie.

$I($`iget` $: prog)\ (a : stack)\ progs\ SP$	$\longrightarrow$	$I\ prog\ (SP[a] : stack)\ progs\ SP$
$I($`iput` $: prog)\ (v : a : stack)\ progs\ SP$	$\longrightarrow$	$I\ prog\ stack\ progs\ SP[a \mapsto v]$

Damit können wir jetzt Programme mit Zuweisungen realisieren:

Beispiel 5.13.6 *Das Programm $x := x + 1$ kann jetzt folgendermaßen erzeugt werden. Allerdings muss vorher von anderer Stelle (Symboltabelle) eine Adresse für x zugewiesen worden sein.*

Befehl	*Stack*	*Anmerkungen*
	$r; 123765$	$SP[123765] = 7$
`push` *0*	$r; 123765; 123765$	
`iget`	$r; 123765; 7$	
`pushK` *1*	$r; 123765; 7; 1$	
$+$	$r; 123765; 8$	
`iput`	r	$SP[123765] = 8$

Weitergehend will man auch mit Objekten im Hauptspeicher wie mit Paaren $\langle a, b\rangle$ umgehen, die aus zwei Adressen a, b (d.h. Zahlen) bestehen und selbst wieder adressierbar sind. D.h. wir nehmen an, dass SP als Einträge entweder ein Paar aus zwei Zahlen hat, oder eine Zahl.

`genpair` Dies Operation nimmt an, dass auf dem Stack oben zwei Adressen stehen. Es wird ein Objekt erzeugt, das diese beiden Adressen enthält, und an einer neuen Adresse steht. Diese wird auf dem Stack abgelegt. Beide Adressen werden vorher gelöscht.

`readpair` Dies Operation nimmt die oberste Element des Stacks als Adresse, löscht diese vom Stack, liest an dieser Adresse im Speicher das Paar und legt die beiden Adressen des Paars auf dem Stack ab.

`getmem` Legt einen Speicherplatz für eine Zahl an und legt die Adresse auf dem Stack ab.

Damit es noch expliziter wird, nehmen wir an, dass die nächste freie Adresse in SP ein weiteres Argument von I ist.

Auch hier geben wir der Vollständigkeit halber die operationale Semantik an:

$I(\texttt{genpair} : prog)\ (b : a : stack)\ progs\ SP\ c$	$\longrightarrow$	$I\ prog\ (c : stack)\ progs\ SP[c \mapsto \langle a, b\rangle]\ (c+1)$
$I(\texttt{readpair} : prog)\ (c : stack)\ progs\ SP\ c$	$\longrightarrow$	$I\ prog\ (b : a : stack)\ progs\ SP[c \mapsto \langle a, b\rangle]\ c$ wobei $SP[c] = \langle a, b\rangle$ ist
$I(\texttt{getmem} : prog)\ stack\ progs\ SP\ c$	$\longrightarrow$	$I\ prog\ (c : stack)\ progs\ SP\ (c+1)$

Damit kann man z.B. Paare und Listen verwalten:

Beispiel 5.13.7 *Die Liste $[1, 2]$ kann man jetzt erzeugen durch:*

Befehl	*Stack*	*Anmerkungen*
	$r;$	
`getmem`	$r; 12348$	
`push` *0*	$r; 12348; 12348$	
`pushK` *1*	$r; 12348; 12348; 1$	
`iput`	$r; 12348$	$SP[12348] := 1$
`getmem`	$r; 12348; 12377$	
`push` *0*	$r; 12348; 12377; 12377$	
`pushK` *2*	$r; 12348; 12377; 12377; 2$	
`iput`	$r; 12348; 12377$	$SP[12377] := 2$
`genpair`	$r; 12399$	$SP[12399] := \langle 12348, 12377\rangle$

Bemerkungen Unsere Stackmaschine ist noch zu abstrakt. Auf einer realen Maschine muss man noch explizitere Angaben machen:

- Die Zahlen müssen von ihrem Speicherbedarf her festgelegt werden (4 byte, 8 byte, ??)
- Die Länge der Adressen muss festgelegt werden.
- Das Programm muss auf einen String abgebildet werden: jeder Befehl auf ein Halbbyte (Byte, o.ä).
- Die symbolischen Marken müssen in Adressen bzgl. des Programms umgewandelt werden.

Hat man diese Längen festgelegt, dann kann man das Programm mittels weiterer Schritte in einen Bitstring, d.h. in ein (Bytecode)-Maschinenprogramm umwandeln:

Jeder Stackmaschinenbefehl wird als ein String kodiert, der am Anfang einen Kode für den Namen des Befehls hat (z.B. push = X'01', pop = X'02', ...). Danach folgt das Argument, bzw. die Argumente, falls der Befehl welche benötigt.

Wenn die Länge der Konstanten und Adressen festgelegt ist und die Kodierung der Befehle als (Halbbyte oder Byte) bekannt ist, kann man schon Teilstrings des endgültigen Programms berechnen. Die Adressen der Sprungbefehle benötigt man als Abstand in Anzahl der Bytes vom Programmanfang. Die Berechnung dieser Adressen muss auf der Grundlage des endgültigen Programms erfolgen. Dies kann man z.B. durch zweifaches Durchgehen des Programms erreichen:

1. die endgültige Form und Länge des Programms wird berechnet, aber die internen Sprungadressen fehlen noch bzw. sind Zeiger in eine Tabelle. Man merkt sich in einer Sprungtabelle die endgültigen Adressen.
2. In diesem Durchgang werden die endgültigen Adressen eingesetzt.

Die Implementierung beliebiger langer Zahlen wie `Integer` in Haskell erfordert eine eigene Implementierung aller arithmetischen Operationen als Programm für die Stackmaschine.

5.13.3 Code-Optimierung durch den Compiler

- Optimierung auf hoher Ebene:
  Im Syntaxbaum oder nach der Zwischencodeerzeugung.
  - Prozedurersetzung (procedure inlining)
  - Schleifentransformation,
  - partielle Auswertung
- Nach Kodeerzeugung
  - Gucklochoptimierung
  - Elimination redundanter Sprünge
  - Umstellungen

Gucklochoptimierungen

Dies sind Optimierungen, die sich lokal bestimmte Sequenzen des erzeugten Codes anschauen und dies bei gleichem Effekt verkürzen. Z.B. kann man die Folge `push i; pop` ersatzlos streichen.

Es ist i.a. nicht möglich, direkt optimalen Code zu erzeugen. Da man schematisch vorgehen muss, gibt es immer Folgen, die lokal unsinning aussehen und redundant sind.

Beispiel 5.13.8 *Für die Funktion $f(x) = x$, oder auch die Identitätsfunktion, wird i.a. folgende Sequenz erzeugt:*

```
push 0; slide 1 1
```

Diese Sequenz ist redundant und kann eliminiert werden.

Diese Optimierungen haben oft einen Satz von Mustern, nach dem sie den erzeugten Code durchforsten, und durch etwas besseres ersetzen. Im allgemeinen kann man durch Gucklochoptimierungen zwar leichte Verbesserungen erzielen, diese haben trotzdem nur den Status von nachträglichen Reparaturen. Zu diesem Zeitpunkt kann die Effizienz nur mit hohem Aufwand verbessert werden.

Prozedurersetzung

aus
`(sqrt (quadratsumme x y))`
wird durch Ersetzung:
`(sqrt (x*x + y*y))`
Dies erspart einen Prozeduraufruf, allerdings kann durch mehrfache Einsetzung der Kode des Gesamtprogramms vergrößert werden! Dies kann im schlimmsten Fall sogar exponentielle Vergrößerung bewirken. In Haskell bleibt bei Prozedureinsetzung die Semantik in jedem Fall erhalten. Bei anderen Programmiersprachen darf man diese Einsetzung nur unter bestimmten Bedingungen machen. Der Einsetzungsprozess darf nicht für Prozeduren bzw. Funktionen verwendet werden, die (auch verschränkt) rekursiv sind, da dann die Einsetzung zur Compilezeit nicht terminiert

Allerdings: Prozedurersetzung bzw. textuelle Ersetzung mittels einfacher Makros erhält nicht in jedem Fall (in Python, Lisp, z.B) die Semantik. Es gibt zwei wesentliche Fälle:

1. Bei Funktionsaufrufen (Z.B. in Python oder Lisp, d.h, in strikten funktionalen Programmiersprachen mit Seiteneffekten) kann sich die Auswertungsreihenfolge verändern, und ebenso das Terminierungsverhalten des Programms.

2. Verdopplung von Seiteneffekten (Zuweisungen, Drucken)

Beispiel 5.13.9

1. *Das Terminierungsverhalten des Programms wird geändert:*

```
def f (x,y):   if y == 0 then x else 0
def g (x):  return  g(x)      # terminiert nicht

f(g(x),1)         # terminiert nicht
```

wird zu

```
if 1 == 0 then g(x) else 0
```

2. *Verdopplung von Seiteneffekten (Zuweisungen, Drucken)*

```
def f (x): x*x
def g(x): print x;return x

f (g x)
```

wird zu

```
g(x) * g(x)
```

Diese Verdopplung tritt ein, wenn ein formaler Parameter der ersetzten Prozedur mehr als einmal im Rumpf vorkommt. Vermeiden kann man diesen Effekt durch ein let-Konstrukt, wobei allerdings der Optimierungseffekt verloren gehen kann.

5.13.4 Partielle Auswertung

Das sind Berechnungen, die man schon zur Compilezeit machen kann. Im Prinzip ist dies auf jede Auswertung oder Berechnung anwendbar, wenn diese nicht von irgendeiner Eingabe abhängt.

statt `10*10 + y*y`
kompiliert man: `100 + y*y`

Eine gute partielle Auswertung verkompliziert den Compiler erheblich. Eine Voraussetzung für partielle Auswertung ist eine festgelegte eindeutige Semantik für das Programm, die nur vom Programmtext abhängt, nicht vom Rechner. Die einfachste Variante ist, dass zur Kompilierzeit ein Interpreter vorhanden ist (d.h. die operationale Semantik ist spezifiziert), der allerdings keine Eingaben anfordern darf. Dieser Interpreter wird dann vom Compiler verwendet, um bestimmte Unterausdrücke auszuwerten. Es besteht hierbei immer die Gefahr, dass dieser Aufruf nicht terminiert, deshalb darf dieser Aufruf i.a. nur eine vorher festgelegte Anzahl von Berechnungschritten ausführen.

Z.B. kann man in einem funktionalen Programm im Prinzip jeden geschlossenen Ausdruck zur Compilezeit auswerten. Eine Verallgemeinerung ist die *Instanziierung*. Damit sind unerwartete Optimierungen möglich:

Beispiel 5.13.10 *Sei die Definition von map*

```
map f [] = []
map f (x:xs) = (f x) : map f xs
```

Dann kann man `map quadrat x` *durch Instanziieren optimieren, indem man den Ausdruck* `(map quadrat)` *in etwas allgemeinerer Form partiell auswertet und eine neue instanziierte Funktionsdefinition erzeugt.*

```
mapq  [] = []
mapq  (x:xs) = (quadrat x) : mapq xs
```

Analog kann man in einem prozeduralen Programm Z.B. ein Java-Programmfragment (hier eine Schleife) der Form:

```
for (int i =1; i > 20; i++) x += 1;
```

ersetzen durch:

```
x += 20
```

5.14 Kode-Erzeugung für Registersatzmaschinen

Die meisten Maschinen-Architekturen sind heutzutage Registersatzmaschinen, die einen Satz von 16-32 Universalregistern besitzen. Üblich sind **Dreiadress-Befehle**

OP DEST, SRC1, SRC2 d.h. DEST:= SRC1 OP SRC2

oder **Zweiadressbefehle**, bei denen ein Quelloperand mit dem Zieloperanden übereinstimmt:

OP DEST, SRC2 d.h. DEST:= DEST OP SRC2

ADD R5, R6 bedeutet z.B. $R5 := R5 + R6$.

Man verwendet RISC (reduced instruction set computer)-Architekturen: Hierbei sind Befehle einheitlich in der kodierten Länge sind. Die Verarbeitung solcher Befehle kann leichter durch Pipelining (Fließbandprinzip) beschleunigt werden als die früher üblichen CISC (complex instruction set computer)-Architekturen. RISC-Architekturen werden besser als **Load/Store**-Architekturen bezeichnet, da sich die Einfachheit des Befehlssatzes darin ausdrückt, dass Speicheradressen Ziel und Quelle nur von Lade- und Speicherbefehlen sein können. Die Operationen wie Addition können ausschließlich mit Registerinhalten ausgeführt werden. Beispielsweise muss die o.a. Operation $SP[a] := SP[a] + R6$ durch drei Befehle realisiert werden:

```
LOAD R0,a  ;  ADD R0,R6  ;  STORE R0,a
```

Maschinenbefehle waren ursprünglich komplexer (CISC) und hatten Befehlsformate, bei denen Hauptspeicheradressen und Register gemischt in Befehlen vorkamen und die Kodierung der Befehle unterschiedlich lang war. Der Nachteil, der sich bei der weitergehenden technischen Entwicklung bemerkbar machte, war die schlechte Parallelisierbarkeit bzw Pipelining von Befehlssequenzen: Die Ausführung einzelner Befehle konnte sehr unterschiedlich lange dauern. Registerbefehle sind schnell, während Hauptspeicherzugriffe länger dauern. Dies führte zum Bremsen der Pipeline. Die RISC-Prinzipien vermeiden häufige und nicht vorhersagbare Wartesituationen, denn es wird schon aus dem Befehls-opcode klar, wie lange er in etwa dauern wird. Zum Beispiel kann eine Folge von arithmetischen Befehlen ohne Load/Store-Befehle dazwischen sehr gut beschleunigt werden.

5.14.1 Codegenerierung für RISC-Registersatzmaschinen

Wir nehmen an, dass Dreiadressbefehle zur Verfügung stehen, mit denen Operationen der Art

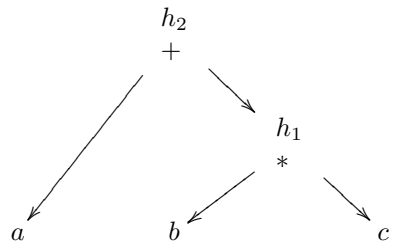
```
DEST:= SRC1 op SRC2
```

ausgeführt werden können (nur auf Registern)

1. Schritt: Einführung von Hilfsvariablen h_i für jeden Knoten im Syntaxbaum. Diese Hilfsvariablen entsprechen den Registern. Dies nennt man auch Überführung in SSA-Form (static single assignment: statische einmalige Zuweisung). Ein Ausdruck wie

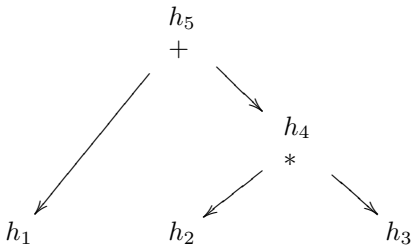
$$a + b * c$$

wird dann wie folgt verarbeitet, wobei für jeden Knoten eine eigene Hilfsvariable benutzt wird:



$$\begin{aligned} h_1 &:= b * c; \\ h_2 &:= h_1 + a; \end{aligned}$$

Bei einer Load/Store-(RISC)-Architektur können nur Register miteinander verknüpft werden. Die Variablen a, b, c müssen vorher in Register geladen werden. Der Ausdruck $a + b * c$ wird daher wie folgt übersetzt:



$$\begin{aligned} h_1 &:= a; \\ h_2 &:= b; \\ h_3 &:= c; \\ h_4 &:= h_2 * h_3; \\ h_5 &:= h_4 + h_1; \end{aligned}$$

2ter Schritt Lebenszeitanalyse der Hilfsvariablen.

Grundblock: linear verarbeitete Sequenz von Anweisungen. Die Lebenszeit einer Hilfsvariablen wird relativ zu Anweisungen angegeben.

Lebenszeit einer Hilfsvariablen:
Eine Variable x heißt *lebendig* (L) für eine Anweisung A in S, wenn beginnend ab der Anweisung A (inklusive), der nächste Zugriff auf x ein Lesezugriff ist. Sonst heißt x tot (T) für A.

Zur Lebenszeitanalyse wird eine Sequenz von Anweisungen **rückwärts** von der letzten zur ersten Anweisung durchlaufen. Beim Analysieren einer Anweisung

$$h_1 := h_2 \; op \; h_3$$

wird

- die Variable h_1 als tot und
- die Variablen h_2 und h_3 als lebendig markiert.

Beispiel 5.14.1 *Ermittelte Lebenszeiten für das Beispiel oben:*

Zuweisung	h_1	h_2	h_3	h_4	h_5
$h_1 := a$	T	T	T	T	T
$h_2 := b$	L	T	T	T	T
$h_3 := c$	L	L	T	T	T
$h_4 := h_2 * h_3$	L	L	L	T	T
$h_5 := h_4 + h_1$	L	T	T	L	T

Damit sind drei Register erforderlich, da das Maximum der L-Einträge pro Zeile $= 3$ *ist.*

Nach einer semantikerhaltenden Umsortierung sind nur noch 2 Register erforderlich.

Zuweisung	h_1	h_2	h_3	h_4	h_5
$h_2 := b$	T	T	T	T	T
$h_3 := c$	T	L	T	T	T
$h_4 := h_2 * h_3$	T	L	L	T	T
$h_1 := a$	T	T	T	L	T
$h_5 := h_4 + h_1$	L	T	T	L	T

Beispiel 5.14.2 *Für eine Load/Store-(RISC)-Architektur wird die Anweisung*

$$y := x - z + w;$$

wie folgt in eine Sequenz mit Hilfsvariablen und angegebenen Lebenszeiten übersetzt:

Zuweisung	h_1	h_2	h_3	h_4	h_5
$h_1 := x$	T	T	T	T	T
$h_2 := z$	L	T	T	T	T
$h_3 := h_1 - h_2$	L	L	T	T	T
$h_4 := w$	T	T	L	T	T
$h_5 := h_3 + h_4$	T	T	L	L	T
$y := h_5$	T	T	T	T	L

3. Schritt: Zuordnung von Registern zu Hilfsvariablen

Zwei Hilfsvariable können gemeinsam in einem Register gehalten werden, wenn sich ihre Lebenszeiten nicht überschneiden. Sie heißen dann **verträglich.** Im Beispiel oben sind h_1, h_3 und h_5 miteinander sowie h_2 und h_4 verträglich.

Im Spezialfall der Sequenz kann man in polynomialer Zeit die Lebenszeitanalyse durchführen: Eine optimale Anzahl von Registern kann man leicht zu finden, indem man die fertige Tabelle von oben nach unten bearbeitet.

Zuweisung	h_1	h_2	h_3	h_4	h_5	R_1	R_2
$h_1 := x$	T	T	T	T	T		
$h_2 := z$	L	T	T	T	T	h_1	
$h_3 := h_1 - h_2$	L	L	T	T	T	h_1	h_2
$h_4 := w$	T	T	L	T	T	h_3	
$h_5 := h_3 + h_4$	T	T	L	L	T	h_3	h_4
$y := h_5$	T	T	T	T	L	h_5	

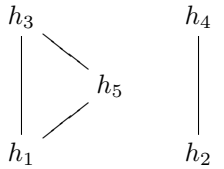
Werden die zugeordneten Register für die Hilfsvariablen eingesetzt, erhält man:

$$\begin{array}{lcl} R_1 & := & x; \\ R_2 & := & z; \\ R_1 & := & R_1 - R_2; \\ R_2 & := & w; \\ R_1 & := & R_1 + R_2; \\ y & := & R_1; \end{array}$$

Im allgemeinen Fall, d.h. wenn die Befehlsfolge keine Sequenz ist, dann wird zur Optimierung der Anzahl der Register ein **Kompatibilitätsgraph** erstellt.

In unserem Fall werden die Hilfsvariablen als Knoten dargestellt; eine Kante $h_i - h_j$ bedeutet, dass zwei Hilfsvariablen miteinander verträglich sind. Die neu eingeführten Hilfsvariablen in verschiedenen Verzweigungen sind unabhängig. Das Problem, eine Zuordnung von Hilfsvariablen zu Registern so zu finden, dass eine minimale Anzahl von Registern benötigt wird, lässt sich als Cliquenpartitionierungsproblem verstehen. Eine Clique ist ein Graph, bei dem es eine Kante von jedem Knoten zu jedem anderen gibt (ein vollständiger Graph). (Eine Clique entspricht dabei einem Register.) Beim Cliquenpartitionierungsproblem versucht man, die Menge der Knoten eines Graphen so zu partitionieren, d.h. in disjunkte Mengen zu unterteilen, dass jede Teilmenge eine Clique bildet und die Anzahl der Cliquen minimal wird.

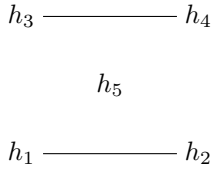
In dem o.a. Beispiel erhält man folgenden einfachen Kompatibilitätsgraphen:



Die beiden Cliquen werden durch die Knotenmengen $\{h_1, h_3, h_5\}$ und $\{h_2, h_4\}$ gebildet.

Die Lösung des Cliquenpartitionierungsproblem hat i.a. exponentiellen Aufwand; es wurden jedoch eine Reihe guter Heuristiken entwickelt, mit denen man zu nicht-optimalen, aber befriedigenden Ergebnissen in akzeptabler Laufzeit kommt.

Ein äquivalente Methode ist das Problem als Färbung eines Graphen mit möglichst wenig Farben zu betrachten: Der **Interferenz-Graph** hat als Knoten die Hilfsvariablen. Zwei Hilfsvariablen h_i und h_j sind verbunden, wenn sich ihre Lebenszeiten überschneiden (sie nicht kompatibel sind)



Man sieht: dieser Graph lässt sich mit zwei Farben einfärben, wobei h_3 und h_4 unterschiedliche Farben bekommen, ebenso h_1 und h_2.

Dieses Verfahren ist günstiger, wenn die Anzahl der inkompatibel-Kanten klein ist. Auch dieses Problem ist $\mathcal{NP}$-vollständig, d.h. die Algorithmen haben im schlechtesten Fall exponentielle Laufzeit.

Im allgemeineren Fall, z.B. wenn Verzweigungen vorkommen, und das Programmfragment ein gerichteter Graph mit einem Anfang (mit oder ohne Rücksprünge) ist, wird ein **Kompatibilitätsgraph** aufgestellt. Dieser ist schwieriger zu berechnen als für lineare Programme.

5.15 LR-Parser, Shift-Reduce-Verfahren

Ein wichtiges Beispiel für die Bottom-Up-Syntaxanalyse sind sogenannte *LR-Parser* (L: die Eingabe wird von links nach rechts verarbeitet; R: eine Rechtsherleitung wird erzeugt). Diese Parser sind häufig so genannte *Shift-Reduce-Parser*. Die Parser sind deterministisch, d.h. ein Zurücksetzen beim Parsen ist nicht erlaubt. Diese Methode lässt sich nicht auf jede Grammatik anwenden, jedoch können viele in der Praxis auftretenden Grammatiken mit dieser Methode geparst werden. Dies ist ein Grund dafür, dass es für diese Parsemethode einige verbreitete so genannte *Parsergeneratoren* gibt, die als Eingabe eine *Parserspezifikationsdatei* nehmen, welche die Grammatik beschreibt, und als Ausgabe einen Parser (als Programm) erzeugen.
Um zu illustrieren, wie diese Parser vorgehen, betrachten wir zunächst ein Beispiel für eine Rechtsherleitung:

Beispiel 5.15.1 *Gegeben sei die CFG:*

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

Betrachte die folgende Rechtsherleitung von $c + b * a$ (unterstrichen ist jeweils das Nichtterminal, dass im nächsten Herleitungsschritt ersetzt wird):

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

Beachte, dass es eine weitere Rechtsherleitung des Ausdrucks gibt.

Definition 5.15.2 *Ein* Handle *(bzgl. einer CFG) ist ein Substring v in einem Wort w über $(T \cup N)^*$, d.h. $w = uvu'$ (für Worte u, u'), so dass*

- $\mathbf{A} \to v$ *eine Produktion ist,*
- *es eine Rechtsherleitung $\sigma \to^* uAu' \to uvu' = w$ gibt d.h. im letzten Schritt $uAu' \to w$ wurde die Produktion $\mathbf{A} \to v$ angewendet. Beachte, dass u' stets nur aus Terminalen besteht, da es sich um eine Rechtsherleitung handelt.*

Als Umkehrung wird ein solcher Handle dann zur Konstruktion einer Rechtsherleitung in umgekehrter Reihenfolge (vom Terminalwort ausgehend zum Startsymbol) verwendet.

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

Beispiel 5.15.3 *(Fortsetzung von Beispiel **??**). Betrachte jetzt den String $c+b*a$ und versuche, eine Rechtsherleitung durch Ersetzen von Handles zu konstruieren (unterstrichen ist jeweils der Handle, der im nächsten Schritt ersetzt wird).*

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

Liest man die Tabelle von unten *nach* oben*, so ergibt sich eine Rechtsherleitung des Wortes $c + b * a$ vom Startsymbol $\mathbf{E}$ ausgehend.*

5.15.1 Schiebe-Reduziere (Shift-Reduce)-Implementierung

Mit diesen Vorbereitungen kann man die sogenannte Shift-Reduce Analysemethode implementieren. Während der Analyse betrachtet man immer die bisher schon gelesene und verarbeitete Eingabe und den Rest des Eingabestroms. Die schon verarbeitete Eingabe liegt dabei auf einem Stack, der den Anfang des bisher hergeleiteten Wortes enthält, so dass der Stack zusammen mit dem Rest der Eingabe das bisher hergeleitete Gesamtwort ergibt.
Die wesentlichen Aktionen der Analyse sind:

Schieben: Lesen des ersten Zeichens der Resteingabe und Verschieben oben auf den Stack

Reduzieren: Ersetzen eines obersten Teils des Stacks, des Handles, mittels einer Produktion, die rückwärts angewendet wird. Die Auswahl der Produktion geschieht anhand des Anfangs der Resteingabe und des Inhalts des Stacks.

Wir verdeutlichen dies am Beispiel. Hierbei markiere \$ das Ende der Eingabe und den Boden des Stacks.

Stack	Eingabe	Aktion
\$	$c + b * a\$$	schiebe
$\$\underline{c}$	$+b * a\$$	reduziere mit $\mathbf{E} ::= c$
$\$\mathbf{E}$	$+b * a\$$	schiebe
$\$\mathbf{E}+$	$b * a\$$	schiebe
$\$\mathbf{E} + \underline{b}$	$* a\$$	reduziere mit $\mathbf{E} ::= b$
$\$\mathbf{E} + \mathbf{E}$	$* a\$$	schiebe
$\$\mathbf{E} + \mathbf{E} *$	$a\$$	schiebe
$\$\mathbf{E} + \mathbf{E} * \underline{a}$	\$	reduziere mit $\mathbf{E} ::= a$
$\$\mathbf{E} + \underline{\mathbf{E} * \mathbf{E}}$	\$	reduziere mit $\mathbf{E} ::= \mathbf{E} * \mathbf{E}$
$\$\underline{\mathbf{E} + \mathbf{E}}$	\$	reduziere mit $\mathbf{E} ::= \mathbf{E} + \mathbf{E}$
$\$\mathbf{E}$	\$	akzeptiere

Wenn man die Tabelle von unten nach oben liest, und Stack und Eingabe aneinanderhängt, sieht man, dass die Reduziere-Schritte gerade eine Rechtsherleitung durchführen.

Allgemein kann ein SR-Parser insgesamt 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 ist und nicht akzeptiert werden kann.

Ein Shift-Reduce-Parser muss eindeutig wissen, was in welcher Situation zu tun ist. Als Information kann er den Inhalt des Stacks und das erste Symbol (für einen LR(1)-Parser, bzw. die ersten k Symbole für einen LR(k)-Parser) 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 nach einer Reduzieraktion: ein Nichtterminal steht oben auf dem Stack. Z.B. $\mathbf{E}+\mathbf{E}$ wird in einem Schritt zu $\mathbf{E}+b$ durch Anwenden der Produktion $\mathbf{E} ::= b$. Das entspricht beim SR-Parser gerade:

Stack	Eingabe	Aktion
$\$\mathbf{E} + b$	...	reduziere mit $\mathbf{E} ::= b$
$\$\mathbf{E} + \mathbf{E}$	...	

Eine Rechtsherleitung kann ein Nichtterminal durch ein Wort ersetzen, das ganz rechts ein Nichtterminal hat. Da beim nächste Herleitungsschritt wieder dieses rechte Nichtterminal ersetzt wird, entspricht dies

mindestens zwei Reduzieraktionen nacheinander. Betrachte z.B. die Herleitung $\mathbf{E} \to \mathbf{E}+\mathbf{E} \to \mathbf{E}+\mathbf{E}*\mathbf{E}$ durch die Produktionen $\mathbf{E} ::= \mathbf{E}+\mathbf{E}$ und $\mathbf{E} ::= \mathbf{E}*\mathbf{E}$. Im SR-Parser werden dann zwei Reduzieraktionen nacheinander durchgeführt:

Stack	Eingabe	Aktion
$\$\mathbf{E}+\mathbf{E}*\mathbf{E}$	...	reduziere mit $\mathbf{E} ::= \mathbf{E}*\mathbf{E}$
$\$\mathbf{E}+\mathbf{E}$	...	reduziere mit $\mathbf{E} ::= \mathbf{E}+\mathbf{E}$
$\$\mathbf{E}$	...	

Wenn das Nichtterminal durch ein Wort mit rechten Terminalen ersetzt wird, dann entspricht das sovielen Schiebeaktionen wie rechts Terminale stehen. Betrachte z.B. die Grammatik $\mathbf{A} ::= \mathbf{A}\mathbf{A}aa$, $\mathbf{A} ::= b$ und die Rechtsherleitung $\mathbf{A} \to \mathbf{A}\mathbf{A}aa \to \mathbf{A}baa$. Dann führt der SR-Parser zwischen den beiden Reduzieraktionen zwei Schiebeaktionen durch, um die beiden a's von der Eingabe auf den Stack zu schieben:

Stack	Eingabe	Aktion
$\$\mathbf{A}b$	$aa\$$	reduziere mit $\mathbf{A} ::= b$
$\$\mathbf{A}\mathbf{A}$	$aa\$$	schiebe
$\$\mathbf{A}\mathbf{A}a$	$a\$$	schiebe
$\$\mathbf{A}\mathbf{A}aa$	$\$$	reduziere mit $\mathbf{A} ::= \mathbf{A}\mathbf{A}aa$
$\$\mathbf{A}$		

Eine ε-Produktion entspricht einer Ersetzung eines leeren Handles durch ein Nichtterminal. Normalerweise werden ε-Produktionen in Shift-Reduce-Parsern vermieden.

Die allgemeine Umsetzung von Grammatiken in einen Shift-Reduce-Parser ist von Hand nicht einfach durchzuführen und ist nicht Gegenstand dieses Skripts. Die Problematik ist die Erzeugung der Tabellen zur Steuerung des Shift-Reduce-Parsers. Für einige spezielle Beispielgrammatiken werden wir einen solchen Parser angeben.

Die Fehlermeldungen, die beim Ablauf eines Shift-Reduce-Parsers auftreten können, sind

- Schiebe-Reduziere-Konflikt: Es kann eine Situation entstehen, in der (aus Sicht der Grammatik) unklar ist, ob geschoben oder reduziert werden soll. Die Tabelle enthält in diesem Fall bereits einen Fehlerausgang.
- Reduziere-Reduziere-Konflikt: Es kann eine Situation entstehen, in der unklar ist, mit welcher Regel reduziert werden soll.

Wir betrachten zunächst ein Beispiel für einen Schiebe-Reduziere-Konflikt. Erlaube die Syntax für eine Programmiersprache sowohl `if-then-else` als auch `if-then` als syntaktische Konstrukte. Eine Grammatik dazu enthalte u.a. die Produktionen

$$\begin{aligned}\mathbf{E} &::= \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{E}\\ \mathbf{E} &::= \texttt{if}\ \mathbf{E}\ \texttt{then}\ \mathbf{E}\ \texttt{else}\ \mathbf{E}\\ \mathbf{E} &::= \texttt{True}\end{aligned}$$

Dann kann der SR-Parser im Fall

Stack	Eingabe	Aktion
... `if` **E** `then` **E**	`else`...	

nicht entscheiden, ob er mit der ersten Produktion reduzieren soll:

Stack	Eingabe	Aktion
... `if` **E** `then` **E**	`else`...	reduziere mit $\mathbf{E} ::= \texttt{if}\ \mathbf{E}+\texttt{then}\ \mathbf{E}$
... **E**	`else`...	

oder, ob er weiter schieben muss:

Stack	Eingabe	Aktion
... `if` **E** `then` **E**	`else`...	schiebe
... `if` **E** `then` **E** `else`	...	

Als Beispiel kann man sich die Verarbeitung von `if True then if True then True else True` anschauen:

Stack	Eingabe	Aktion
$	`if True then if True then True else True`$	schiebe
$`if`	`True then if True then True else True`$	schiebe
$`if True`	`then if True then True else True`$	reduziere mit **E** ::= `True`
$`if` **E**	`then if True then True else True`$	schiebe
$`if` **E** `then`	`if True then True else True`$	schiebe
$`if` **E** `then if`	`True then True else True`$	schiebe
$`if` **E** `then if True`	`then True else True`$	reduziere mit **E** ::= `True`
$`if` **E** `then if` **E**	`then True else True`$	schiebe
$`if` **E** `then if` **E** `then`	`True else True`$	schiebe
$`if` **E** `then if` **E** `then True`	`else True`$	reduziere mit **E** ::= `True`
$`if` **E** `then if` **E** `then` **E**	`else True`$	

Schiebe-Reduziere-Konflikt

Ein Reduziere-Reduziere-Konflikt tritt auf, wenn der Parser zwar weiß, dass er reduzieren muss, aber mehrere Produktionen passen und es daher unklar ist, mit welcher Produktion reduziert werden soll. Betrachte als einfaches Beispiel eine Sprache, die Haskell's Listensyntax sowohl für Listen als auch für Tupel verwendet (wobei *var* das Token für Variablennamen sei):

E ::= **Tupel** | **Liste** | *var*
Tupel ::= [**Elemente**]
Liste ::= [**Elemente**] | []
Elemente ::= **E** | **E**, **Elemente**

Im Zustand

Stack	Eingabe	Aktion
$[**Elemente**]	...	

tritt ein Reduziere-Reduziere-Konflikt auf, da sowohl mit der Produktion **Tupel** → [**Elemente**] als auch mit der Produktion **Liste** → [**Elemente**] reduziert werden könnte.

Beide Arten von Konflikten können auch schon während der Erzeugung eines SR-Parsers durch einen Parsergenerator entdeckt werden. Dieser generiert dann eine Fehlermeldung oder eine Warnung und gibt Hinweise, wo die entsprechenden Konflikte auftreten. Oft versuchen die Parsergeneratoren die Konflikte aufzulösen, indem sie einfach eine der Regeln bevorzugen (z.B. bei einem Schiebe-Reduziere-Konflikt stets Schieben). Man muss sich klarmachen, dass in diesem Fall der erzeugte Parser nicht mehr zur Grammatik passt, und eine *andere* Sprache erkennt. Als Daumenregel kann man sich merken, dass Reduziere-Reduziere-Konflikte auf jeden Fall zu vermeiden sind (indem man z.B. die Syntax abändert (verwende z.B. runde Klammern für Tupel und eckige Klammern für Listen)). Schiebe-Reduziere-Konflikte werden manchmal toleriert, z.B. würde für obiges `if-then`-Beispiel stets geschoben, was der Klammerung `if True then` (`if True then True else True`) entspricht.

5.15.2 Operatorgrammatiken, Operator-Prioritäts-Syntaxanalyse

Diese Grammatiken erlauben die einfache Konstruktion eines Shift-Reduce Parsers, wenn man weitere Angaben zu den Eigenschaften der Operatoren wie $+, -, *$ usw. macht.

Definition 5.15.4 *Eine* Operatorgrammatik *hat als kennzeichnende Merkmale, dass es keine ε-Produktionen gibt und auch keine rechte Seite einer Produktion mit direkt benachbarten Nichtterminalen.*

Die Terminale sind unterteilt in Operatoren und Klammern, die wesentlichen Nichtterminale entsprechen Ausdrücken, oder Bezeichnern.

Beispiel 5.15.5 *Die Grammatik*

$$\mathbf{E} ::= \mathbf{E} + \mathbf{E} \mid -\mathbf{E} \mid \mathbf{E} - \mathbf{E} \mid \mathbf{E} * \mathbf{E} \mid \mathbf{E}/\mathbf{E} \mid (\mathbf{E}) \mid \mathbf{E} \hat{} \mathbf{E} \mid \mathbf{E}! \mid \mathbf{Id}$$

ist eine Operatorgrammatik. Hierbei steht **Id** *(Identifier) für ein Nichtterminal, aus dem man beispielsweise Konstanten, oder Variablen herleiten kann.*
Beachte, dass diese Grammatik mehrdeutig ist.
Die Grammatik

$$\begin{array}{ll} \mathbf{E} & ::= \mathbf{E}\ \mathbf{BOP}\ \mathbf{E} \mid \mathbf{Id} \\ \mathbf{BOP} & ::= + \mid * \mid - \mid / \end{array}$$

ist keine *Operatorgrammatik, da in* $\mathbf{E} \to \mathbf{E}\ \mathbf{BOP}\ \mathbf{E}$ *auf der rechten Seite Nichtterminale ohne Operator dazwischen vorkommen.*

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: welchen Ausdruck (welche Ausdrücke) bindet der Operator, bezogen auf seine Position?

Prioritäten von Operatoren: welcher Operator bindet stärker? Man verwendet i.a. natürliche Zahlen, um die Stärke der Priorität anzugeben.

Assoziativität: Ist der Operator rechtsassoziativ oder linksassoziativ, oder nicht assoziativ? Z.B. $a + b + c$ kann als $a + (b + c)$ (rechtsassoziativ) oder als $(a + b) + c$ (linksassoziativ) geklammert werden.

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

Beispiel 5.15.6 $1 + 2!\hat{}3$ *entspricht* $1 + ((2!)\hat{}3)$
$1 - 3 - 5 * 6!$ *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 5.15.7 *Als Beispiel wollen wir die Grammatik konstruieren, die zu einem Postfix-Operator ! mit hoher Priorität, dem einstelligen − mit geringerer Priorität und dem Zeichen + mit geringster Priorität gehört.* **Zahl** *sei das Nichtterminal für Zahlenkonstanten.*

$$\begin{array}{lll} \mathbf{E} & ::= & \mathbf{PlusE} \\ \mathbf{FakE} & ::= & \mathbf{Zahl} \mid \mathbf{FakE}\ \mathtt{!} \mid (\mathbf{PlusE}) \\ \mathbf{MinE} & ::= & \mathbf{FakE} \mid \mathtt{-}\ \mathbf{MinE} \\ \mathbf{PlusE} & ::= & \mathbf{MinE} \mid \mathbf{PlusE}\ \mathtt{+}\ \mathbf{FakE} \end{array}$$

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.

5.15.3 Vorgehen bei der Shift-Reduce Syntaxanalyse für Operatorgrammatiken

Für Operatorgrammatiken kann man einen Shift-Reduce Parser relativ einfach von Hand programmieren.
Eine (nicht vollständige) Liste von möglichen Aktionen ist:

1. Die Eingabe wird von links nach rechts abgearbeitet. Auf dem Stack merkt man sich in der Reihenfolge der Eingabe:
   - die bisherigen Operatoren,
   - die offenen Klammern und
   - Bezeichner und erkannte Ausdrücke (d.h. Herleitungsbäume)
2. Wenn ein Bezeichner in der Eingabe ist, dann folgt eine Schiebe-Aktion. In gewissen Fällen kann man danach reduzieren, Z.B., wenn −;E auf dem Stack steht.
3. Wenn Tokenstrom zu Ende, dann reduzieren, bzw. Fehler, wenn kein Reduzieren möglich.
4. Bei „(“ , dann schieben
5. Bei „)“: Wenn der Stack noch offene Operatoren enthält, dann reduzieren.

   Am Ende muss auf dem Stack „(“ ; E stehen: Der Klammerausdruck (E) wird erkannt, und der Syntaxbaum für E bleibt auf dem Stack.
6. Unter der Annahme, dass es nur binäre Operatoren gibt, gilt:
   Wenn ein Operator op in der Eingabe: Suche letzten Operator auf dem Stack.
   - Wenn dieser größere Priorität hat, dann reduzieren.
   - Wenn dieser gleiche Priorität hat und beide linksassoziativ, dann ebenfalls auf dem Stack reduzieren.
   - Wenn gleiche Priorität und beide rechtsassoziativ, dann schieben.
   - Wenn niedrigere Priorität, dann schieben.

Wenn es Infix und Postfix-Operatoren gibt, dann muss man noch Unterfälle betrachten (siehe das Programm `shr-parser.hs`, allerdings ist auch das kein allgemeiner Fall.)

Beispiel 5.15.8 *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*	*! $*	*R*
*$E - E*E*	*! $*	*S*
*$E - E*6!*	*$*	*R*
*$E - E*E*	*$*	*R*
$E - E	*$*	*R*
$E	*$*	

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

Bemerkung 5.15.9 *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 Präfix 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.*
- *Die Semantik kann direkt aus dem Syntaxbaum abgelesen werden.*
- *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, d.h. als Argument zu verwenden. Ebenso partielle Anwendungen wie* `(1 +)`*. Diese Erweiterungen sind unproblematisch.*

5.15.4 SLR(1)-Parser

Im Gegensatz zu dem oben vorgestellten Verfahren für Operatorgrammatiken, arbeiten SLR(1)-Parser (S steht dabei für Simple) leicht anders, sie können aber wesentlich mehr Grammatiken verarbeiten. Die Konstruktion dieser Parser ist jedoch ziemlich aufwändig, so dass wir diese nicht besprechen. Dies ist auch nicht unbedingt nötig, da Parsergeneratoren solche SLR(1)-Parser generieren können. Wir erläutern jedoch, wie die erstellten Parser vorgehen.

Ein SLR(1)-Parser besteht aus:

- einer endliche Menge von Zuständen, die üblicherweise durch natürliche Zahlen ausgedrückt werden, wobei ein Zustand als *Startzustand* gekennzeichnet ist.
- einem Stack der Form $s_1 X_1 s_2 X_2 \dots s_n X_n s_{n+1}$. Hierbei sind X_i Terminale oder Nichtterminale der Grammatik und s_i sind Zustände des Parsers
- dem Eingabestrom (von Token)
- zwei Steuerungstabellen (bzw. Funktionen):
  - Die *Aktionstabelle* legt für jeden Zustand s_i und dem ersten Symbol a_i des Eingabestroms eine Aktion $aktion(s_i, a_i)$ (und zusätzlich für den Fall der leeren Eingabe $aktion(s_i, \$)$) fest. Aktionen können dabei sein:

  - ∗ Schiebe und gehe zu Zustand s_k (nicht möglich für den Fall der leeren Eingabe!)
  - ∗ Reduziere (mit Angabe der Produktion der Grammatik)
  - ∗ Akzeptiere
  - ∗ Fehler
- Die *Goto-Tabelle* legt für einen Zustand s_i und ein Nichtterminal A einen Nachfolgezustand $goto(s_i, A)$ fest. Die Funktion ist partiell, d.h. sie muss nicht für jeden Zustand und jedes Nichtterminal festgelegt werden.

Wir beschreiben den Ablauf des Parsers:

Der Parser startet mit dem Startzustand als einzigem Eintrag auf dem Stack und dem Eingabestrom. Anschließend wird iteriert:

Sei $s_1X_1s_2X_2 \dots s_nX_ns_{n+1}$ der Stack und w der Eingabestrom Wenn der Eingabestrom nicht leer ist, berechne $aktion(s_{n+1}, a)$ wobei a das erste Symbol der Eingabe w ist. Ansonsten berechne $aktion(s_{n+1}, \$)$.

- Wenn die berechnete Aktion ist „akzeptiere“, dann stoppe und akzeptiere.
- Wenn die berechnete Aktion ist „Fehler“, dann stoppe und melde Fehler.
- Wenn die berechnete Aktion ist „schiebe und gehe zu Zustand s_j“, dann schiebe erst a und lege anschließend s_j auf den Stack und iteriere weiter. D.h. der Stack hat danach die Form $s_1X_1s_2X_2 \dots s_nX_ns_{n+1}as_j$.
- Wenn die berechnete Aktion ist „reduziere mit Produktion $E \to X_1 \dots X_m$“. Dann entferne $2 * m$ Einträge von oben auf dem Stack. Anschließend muss ein Zustand s_i als oberstes Symbol auf dem Stack liegen. Sei $goto(s_i, E) = s_j$. Lege erst E und anschließend s_j auf den Stack und iteriere weiter.

Beachte: Durch das Merken der Zustände auf dem Stack, kann bei einer Reduziere-Aktion mit unterschiedlichen Zuständen fortgefahren werden (das bestimmt die Goto-Tabelle!).

Der schwierige Teil bei dieser Methode ist das Erstellen der richtigen Aktions- und Goto-Tabellen. Wir verzichten auf diesen Teil. Wir betrachten ein Beispiel:

Wir betrachten die folgende Grammatik mit Startsymbol **E** (wobei Z ein Token für eine Zahl sei):

$$\mathbf{E} ::= \mathbf{E} + \mathbf{E} \mid (\mathbf{E}) \mid Z$$

Zusätzlich nehmen wir an, dass + als linksassoziativ bekannt ist.

Ein Parsergenerator kann daraus die folgenden Aktions- und Goto-Tabellen erstellen, wobei es 10 Zustände 0,1,. . . , 9 gibt und 0 der Startzustand ist.

Zustand	Aktion					Goto
Symbol	\$	Z	(	)	+	**E**
0		(s,3)	(s,4)			5
1		(s,3)	(s,4)			2
2					(s,6)	
3	(r,$\mathbf{E} \to Z$)			(r,$\mathbf{E} \to Z$)	(r,$\mathbf{E} \to Z$)	
4		(s,3)	(s,4)			7
5	akzeptiere				(s,6)	
6		(s,3)	(s,4)			9
7				(s,8)	(s,6)	
8	(r,$\mathbf{E} \to (\mathbf{E})$)			(r,$\mathbf{E} \to (\mathbf{E})$)	(r,$\mathbf{E} \to (\mathbf{E})$)	
9	(r,$\mathbf{E} \to \mathbf{E} + \mathbf{E}$)			(r,$\mathbf{E} \to \mathbf{E} + \mathbf{E}$)	(r,$\mathbf{E} \to \mathbf{E} + \mathbf{E}$)	

Leere Einträge in der Aktionstabelle beschreiben die Fälle, in denen ein Fehler auftritt. Einträge (s,i) bedeuten „Schiebe und lege Zustand i danach auf den Stack“. Einträge der Form (r,Produktion) bedeuten „Reduziere mit der entsprechenden Produktion“.

Wir betrachten die Abarbeitung des Parsers bei Eingabe $a + b + (c + d)$ (dabei seien a, b, c, d Zahlen, die wie Z behandelt werden):

Stack	Eingabe	Aktion	Bemerkungen
$\$0$	$a+b+(c+d)\$$	(s,3)	
$\$0, \underline{a, 3}$	$+b+(c+d)\$$	(r,$\mathbf{E} \to Z$)	2 Symbole entfernen, $goto(0, \mathbf{E}) = 5$. Daher **E**,5 auf den Stack legen
$\$0, \mathbf{E}, 5$	$+b+(c+d)\$$	(s,6)	
$\$0, \mathbf{E}, 5, +, 6$	$b+(c+d)\$$	(s,3)	
$\$0, \mathbf{E}, 5, +, 6, \underline{b, 3}$	$+(c+d)\$$	(r,$\mathbf{E} \to Z$)	2 Symbole entfernen, $goto(6, \mathbf{E}) = 9$. Daher **E**,9 auf den Stack legen
$\$0, \underline{\mathbf{E}, 5, +, 6, \mathbf{E}, 9}$	$+(c+d)\$$	(r,$\mathbf{E} \to \mathbf{E}+\mathbf{E}$)	6 Symbole entfernen, $goto(0, \mathbf{E}) = 5$. Daher **E**,5 auf den Stack legen
$\$0, \mathbf{E}, 5$	$+(c+d)\$$	(s,6)	
$\$0, \mathbf{E}, 5, +, 6$	$(c+d)\$$	(s,4)	
$\$0, \mathbf{E}, 5, +, 6, (, 4$	$c+d)\$$	(s,3)	
$\$0, \mathbf{E}, 5, +, 6, (, 4, \underline{c, 3}$	$+d)\$$	(r,$\mathbf{E} \to Z$)	2 Symbole entfernen, $goto(4, \mathbf{E}) = 7$. Daher **E**,7 auf den Stack legen
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7$	$+d)\$$	(s,6)	
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7, +, 6$	$d)\$$	(s,3)	
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7, +, 6, \underline{d, 3}$	$)\$$	(r,$\mathbf{E} \to Z$)	2 Symbole entfernen, $goto(6, \mathbf{E}) = 9$. Daher **E**,9 auf den Stack legen
$\$0, \mathbf{E}, 5, +, 6, (, 4, \underline{\mathbf{E}, 7, +, 6, \mathbf{E}, 9}$	$)\$$	(r,$\mathbf{E} \to \mathbf{E}+\mathbf{E}$)	6 Symbole entfernen, $goto(4, \mathbf{E}) = 7$. Daher **E**,7 auf den Stack legen
$\$0, \mathbf{E}, 5, +, 6, (, 4, \mathbf{E}, 7$	$)\$$	(s,8)	
$\$0, \mathbf{E}, 5, +, 6, \underline{(, 4, \mathbf{E}, 7,), 8}$	$\$$	(r,$\mathbf{E} \to (\mathbf{E})$)	6 Symbole entfernen, $goto(6, \mathbf{E}) = 9$. Daher **E**,9 auf den Stack legen
$\$0, \underline{\mathbf{E}, 5, +, 6, \mathbf{E}, 9}$	$\$$	(r,$\mathbf{E} \to \mathbf{E}+\mathbf{E}$)	6 Symbole entfernen, $goto(0, \mathbf{E}) = 5$. Daher **E**,5 auf den Stack legen
$\$0, \mathbf{E}, 5$	$\$$	akzeptiere	

Beachte, dass man in der echten Implementierung die Terminale und Nichtterminale nicht auf dem Stack speichern muss, denn es reicht aus lediglich die Zustände zu speichern (und beim Reduzieren nur die Hälfte an Symbolen zu entfernen).

5.15.5 Parsergeneratoren am Beispiel Happy

Ein Parsergenerator ist ein Werkzeug (Programm), das dabei hilft einen Parser zu erzeugen. D.h. anstatt den Parser komplett selbst von Hand zu Programmieren, wird dieser automatisch anhand einer sogenannten Parserspezifikation erzeugt. Es gibt verschiedene Parsergeneratoren für verschiedene Programmiersprachen. Z.B. erzeugt Yacc[2] einen Parser in der Programmiersprache C, sodass sie Ausgabe des Parsers direkt als C-Datenstruktur verwendet werden kann. Auch für Haskell gibt es Parsergeneratoren. Der bekannteste und meist-verwendete ist „Happy“[3]. Das Format der Parserspezifikation wie auch die Technik ist dabei stark an Yacc angelehnt.

Generell haben Parsergenerator im Wesentlichen als Eingabe, eine kontextfreie Grammatik und weitere Festlegungen wie die Angaben zu den Operatoren (Assoziativitäten und Prioritäten). Happy (wie auch Yacc für C) erzeugt einen SLR(1)-Parser für die eingegebene Grammatik.

5.15.6 Aufbau eines Happy-Skripts

In diesem Abschnitt werden wir den Aufbau einer Happy-Parserspezifikation beschreiben. Dafür betrachten wir die folgende Grammatik für arithmetische Ausdrücke:

[2] Yacc steht für „yet another compiler compiler“
[3] Verfügbar unter `http://haskell.org/happy`

$$
\begin{array}{rcl}
\mathbf{E} & ::= & \mathbf{E} + \mathbf{E} \\
& | & \mathbf{E} - \mathbf{E} \\
& | & \mathbf{E} * \mathbf{E} \\
& | & \mathbf{E} \,/\, \mathbf{E} \\
& | & (\ \mathbf{E}\ ) \\
& | & \textit{zahl}
\end{array}
$$

Hierbei sei *zahl* das Token für eine Zahl. Ein Lexer für diese Sprache ist schnell von Hand programmiert. Wir benutzen hierfür als Ausgabe den Typ `Token` definiert als:

```
data Token = TokenInt Int
           | TokenSymbol Char
 deriving(Show)
```

Die Funktion `lexer` führt nun die lexikalische Analyse durch und überführt einen Eingabestring in eine Liste von Token. Diese kann programmiert werden als:

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
 | c `elem` ['+','-','*','/','(',')'] =
                 (TokenSymbol c) : lexer cs
 | isSpace c = lexer cs
 | isDigit c = lexNum (c:cs)
 | otherwise = error ("parse error, can't lex symbol " ++ show c)

lexNum cs = TokenInt (read num) : lexer rest
              where (num,rest) = span isDigit cs
```

Der Aufbau einer Happy-Parserspezifikation (die Dateiendung muss `.y` lauten) ist von der Form

Modulkopf (optional)
Parserdirektiven
`%%`
Grammatik
Modulschluss (optional)

Nehmen wir an, eine solche Spezifikation steht in der Datei `Parser.y`, dann erzeugt der Aufruf `happy Parser.y` einen Shift-Reduce-Parser in der Datei `Parser.hs`.

Der erste (optionale) Teil – der Modulkopf – ist ein Block Haskell-Code, der von geschweiften Klammern umschlossen wird. Dieser Block wird unverändert an den Anfang der durch `Happy` generierten Datei gesetzt. Für gewöhnlich stehen hier der Modulkopf, Typdeklarationen, `import`-Befehle usw.

Für unser Beispiel könnten wir in den Modulkopf schreiben:

```
{
module Parser where
import Data.Char
}
```

Der nächste Teil enthält verschiedene Direktiven, die Happy für eine korrekte Funktionsweise unbedingt benötigt:

- `%name` *NAME* bezeichnet den Namen der Parserfunktion. Unter diesem Namen kann der Parser später aufgerufen werden.

- `%tokentype` { *TYPE* } Dies ist der Ausgabetyp des Lexers und damit der Eingabetyp des Parsers.
- `%token` *MATCHLIST* Hier werden den Token, die vom Lexer erzeugt wurden, die *Terminals* zugewiesen, die in der BNF verwendet werden.

Für unser Beispiel vergeben wir den Namen `parser` für die Parserfunktion, der Tokentyp ist `Token` und für die Zuordnung zwischen Token und Terminalen der Grammatik benutzen wir die üblichen Symbole.

```
%name parser
%tokentype { Token }
%token
        int         { TokenInt $$ }
        '+'         { TokenSymbol '+' }
        '-'         { TokenSymbol '-' }
        '*'         { TokenSymbol '*' }
        '/'         { TokenSymbol '/' }
        '('         { TokenSymbol '(' }
        ')'         { TokenSymbol ')' }
```

Für die Zuweisung der Terminals an die Tokens gilt: Links stehen die Terminals, rechts in geschweiften Klammern die Token.

Das Symbol $$ ist ein Platzhalter, das den Wert des Tokens repräsentiert. Normalerweise ist der Wert eines Tokens der Token selbst, mit $$ wird ein Teil des Tokens als Wert spezifiziert. Im Beispiel ist der Wert des Tokens `TokenInt` *zahl* die Zahl.

Es schließt sich der Grammatikteil an (vom zweiten Teil durch ein `%%` getrennt), in dem in einer BNF ähnlichen Notation die Syntax, wie man sie sich zuvor überlegt hat, aufgeschrieben wird. Hinter jede Regel wird in geschweiften Klammern eine so genannte Aktion geschrieben.

```
%%
E :: { Expr }
E : E '+' E         { Plus  $1 $3 }
  | E '-' E         { Minus $1 $3 }
  | E '*' E         { Times $1 $3 }
  | E '/' E         { Div $1 $3 }
  | '(' E ')'       { $2 }
  | int             { Number $1 }
```

In der ersten Zeile wird dabei der Ausgabetyp definiert, den der Parser für Worte erzeugt, die vom Nichtterminal `E` erzeugt werden. Im Beispiel haben wir den Typ `Expr` verwendet, der einen Syntaxbaum für arithmetische Ausdrücke darstellt und definiert ist als:

```
data Expr   = Plus    Expr Expr
            | Minus   Expr Expr
            | Times   Expr Expr
            | Div     Expr Expr
            | Number  Int
  deriving(Show)
```

Die weiteren Zeilen beschreiben die eigentliche Grammatik (anstelle von ::= wird dabei `:` verwendet)

Hinter jeder Produktion steht dabei in geschweiften Klammern eine sogenannte Aktion. Dies ist ein Stück Haskell-Code, der angibt, was bei erfolgreichem Parsen (Reduzieren) mit dem Ergebnis geschehen soll. Der Code muss zusichern, dass tatsächlich ein Objekt vom Typ `Expr` erzeugt wird. Mittels `$i` wird dabei auf den Wert des i-ten Terminals bzw. Nonterminals zugegriffen. Der Wert eines Terminals ist dabei normalerweise das Terminal selbst (beachte für `int` ist es die Zahl selbst, da wir es oben so definiert haben). Wir betrachten als Beispiel die Produktion `E` → `E '*' E { Times $1 $3 }`: Die Aktion besagt, dass aus den Parseergebnissen

für das erste `E` (referenziert durch $1) und das zweite `E` (referenziert durch $3, da es das dritte Symbol auf der rechten Seite der Produktion ist) ein Objekt mit dem Konstruktor `Times` konstruiert werden soll.

Durch die Aktionen ist der Parser daher nicht nicht nur ein reiner Syntax-Überprüfer, sondern er bearbeitet die Ausgabe gleichzeitig,

Der vierte Teil eines `Happy`-Skripts ist wieder ein in geschweifte Klammern gesetzter Block mit Haskell-Code, welcher unverändert ans Ende der erzeugten Datei gesetzt wird. Hier sollte zumindest die Funktion `happyError` stehen, welche im Fall eines Syntax-Fehlers von der Parser-Funktion automatisch angesprungen wird (damit dies funktioniert, darf für diese Funktion kein anderer Name verwendet werden.) Die Funktion `happyError` hat den Typ `[Token] -> a`, d.h. sie erhält den Tokenstrom (ab der Stelle, wo der Parser einen Fehler findet) und sollte dann mithilfe der `error`-Funktion eine Fehlermeldung erzeugen.

```
happyError :: [Token] -> a
happyError [] = error "parse error: unerwartetes Ende"
happyError xs = error ("parse error:" ++ show xs)
```

Für unser Beispiel können wir im Modulschluss den Typ `Expr` definieren (dieser muss auch irgendwo definiert werden), und den Lexer samt Tokendatentyp einfügen.

Abbildung **??** zeigt die gesamte Parserspezifikation.

Ruft man nun `happy Parser1.y` auf, so erstellt Happy die Datei `Parser1.hs` und man kann diese im ghci laden und mit `parse` verschiedene Eingaben parsen. Allerdings gibt happy beim Erstellen eine Warnung aus:

```
> happy Parser1.y
shift/reduce conflicts:  16
```

Happy hat in diesem Fall somit festgestellt, dass die Grammatik 16 Schiebe-Reduziere-Konflikte hat. Zur Fehlersuche kann man Happy mit der Option `-i`*Infodateiname* aufrufen. Dann erzeugt Happy in der Datei *Infodateiname* Informationen zum Parser, die genau der Aktions- und Goto-Tabelle entsprechen. Abbildung **??** zeigt die Ausgabe dieser Datei für den Aufruf `happy -iInfoDatei Parser1.y`.

Wer möchte kann dort die Schiebe-Reduziere-Konflikte nachvollziehen. Sie entstehen jedoch dadurch, dass die Grammatik mehrdeutig ist und wir keine Prioritäten und Assoziativitäten angegeben haben. Dies ist jedoch auch in Happy möglich. Direkt vor `%%` können diese eingefügt werden.

- `%left` *Terminal(e)* legt fest, dass diese Terminale links-assoziativ sind (d.h. ein Ausdruck $a \otimes b \otimes c$ wird als $(a \otimes b) \otimes c$ aufgefasst).
- `%right` *Terminal(e)* legt fest, dass diese Terminale rechts-assoziativ sind (d.h. ein Ausdruck $a \otimes b \otimes c$ wird als $a \otimes (b \otimes c)$ aufgefasst).
- `%nonassoc` *Terminal(e)* legt fest, dass diese Terminale nicht assoziativ sind (d.h. ein Ausdruck $a \otimes b \otimes c$ kann nicht geparst werden und es tritt ein Fehler auf)

Die Präzedenz der Terminale gegenüber den anderen Terminalen wird durch die Reihenfolge `%left`, `%right` und `%nonassoc` Direktiven festgelegt, wobei „früher“ „weniger Präzedenz“ bedeutet. Nach dem Einfügen der Zeilen

```
%left '+' '-'
%left '*' '/'
```

direkt vor `%%` (siehe Abbildung **??**) hat der Parser keine Konflikte mehr und parst arithmetische Ausdrücke entsprechend der üblichen geltenden Konventionen (Punkt vor Strich usw.).

```
{
module Parser where
import Data.Char
}
%name parser
%tokentype { Token }
%token
        int        { TokenInt $$ }
        '+'        { TokenSymbol '+' }
        '-'        { TokenSymbol '-' }
        '*'        { TokenSymbol '*' }
        '/'        { TokenSymbol '/' }
        '('        { TokenSymbol '(' }
        ')'        { TokenSymbol ')' }
%%
E :: { Expr }
E : E '+' E         { Plus  $1 $3 }
  | E '-' E         { Minus $1 $3 }
  | E '*' E         { Times $1 $3 }
  | E '/' E         { Div $1 $3 }
  | '(' E ')'       { $2 }
  | int             { Number $1 }

{

happyError :: [Token] -> a
happyError [] = error "parse error: unerwartetes Ende"
happyError xs = error ("parse error:" ++ show xs)


data Token = TokenInt Int
           | TokenSymbol Char
 deriving(Show)

lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
 | c `elem` ['+','-','*','/','(',')'] =
                 (TokenSymbol c) : lexer cs
 | isSpace c = lexer cs
 | isDigit c = lexNum (c:cs)
 | otherwise = error ("parse error, can't lex symbol " ++ show c)

lexNum cs = TokenInt (read num) : lexer rest
              where (num,rest) = span isDigit cs

data Expr   = Plus    Expr Expr
            | Minus   Expr Expr
            | Times   Expr Expr
            | Div     Expr Expr
            | Number  Int
  deriving(Show)
}
```

Abbildung 5.2: Happy-Parserspezifikation für arithmetische Ausdrücke (Parser1.hs)

```
-----------------------------------------------
Info file generated by Happy from Parser1.y
-----------------------------------------------
state 12 contains 4 shift/reduce conflicts.
state 13 contains 4 shift/reduce conflicts.
state 14 contains 4 shift/reduce conflicts.
state 15 contains 4 shift/reduce conflicts.
-----------------------------------------------
Grammar
-----------------------------------------------
        %start_parser -> E              (0)
        E -> E '+' E                    (1)
        E -> E '-' E                    (2)
        E -> E '*' E                    (3)
        E -> E '/' E                    (4)
        E -> '(' E ')'                  (5)
        E -> int                        (6)
-----------------------------------------------
Terminals
-----------------------------------------------
        int             { TokenInt $$ }
        '+'             { TokenSymbol '+' }
        '-'             { TokenSymbol '-' }
        '*'             { TokenSymbol '*' }
        '/'             { TokenSymbol '/' }
        '('             { TokenSymbol '(' }
        ')'             { TokenSymbol ')' }
-----------------------------------------------
Non-terminals
-----------------------------------------------
        %start_parser   rule  0
        E               rules 1, 2, 3, 4, 5, 6
-----------------------------------------------
States
-----------------------------------------------
State 0 int             shift, and enter state 3
        '('             shift, and enter state 4

        E               goto state 5

State 1 int             shift, and enter state 3
        '('             shift, and enter state 4

        E               goto state 2

State 2 '+'             shift, and enter state 6
        '-'             shift, and enter state 7
        '*'             shift, and enter state 8
        '/'             shift, and enter state 9

State 3 '+'             reduce using rule 6
        '-'             reduce using rule 6
        '*'             reduce using rule 6
        '/'             reduce using rule 6
        ')'             reduce using rule 6
        %eof            reduce using rule 6

State 4 int             shift, and enter state 3
        '('             shift, and enter state 4

        E               goto state 10

State 5 '+'             shift, and enter state 6
        '-'             shift, and enter state 7
        '*'             shift, and enter state 8
        '/'             shift, and enter state 9
        %eof            accept

State 6 int             shift, and enter state 3
        '('             shift, and enter state 4

        E               goto state 15

State 7 int             shift, and enter state 3
        '('             shift, and enter state 4

        E               goto state 14

State 8 int             shift, and enter state 3
        '('             shift, and enter state 4

        E               goto state 13

State 9 int             shift, and enter state 3
        '('             shift, and enter state 4

        E               goto state 12

State 10 '+'            shift, and enter state 6
         '-'            shift, and enter state 7
         '*'            shift, and enter state 8
         '/'            shift, and enter state 9
         ')'            shift, and enter state 11

State 11 '+'            reduce using rule 5
         '-'            reduce using rule 5
         '*'            reduce using rule 5
         '/'            reduce using rule 5
         ')'            reduce using rule 5
         %eof           reduce using rule 5

State 12 '+'            shift, and enter state 6
                        (reduce using rule 4)
         '-'            shift, and enter state 7
                        (reduce using rule 4)
         '*'            shift, and enter state 8
                        (reduce using rule 4)
         '/'            shift, and enter state 9
                        (reduce using rule 4)
         ')'            reduce using rule 4
         %eof           reduce using rule 4

State 13 '+'            shift, and enter state 6
                        (reduce using rule 3)
         '-'            shift, and enter state 7
                        (reduce using rule 3)
         '*'            shift, and enter state 8
                        (reduce using rule 3)
         '/'            shift, and enter state 9
                        (reduce using rule 3)
         ')'            reduce using rule 3
         %eof           reduce using rule 3

State 14 '+'            shift, and enter state 6
                        (reduce using rule 2)
         '-'            shift, and enter state 7
                        (reduce using rule 2)
         '*'            shift, and enter state 8
                        (reduce using rule 2)
         '/'            shift, and enter state 9
                        (reduce using rule 2)
         ')'            reduce using rule 2
         %eof           reduce using rule 2

State 15 '+'            shift, and enter state 6
                        (reduce using rule 1)
         '-'            shift, and enter state 7
                        (reduce using rule 1)
         '*'            shift, and enter state 8
                        (reduce using rule 1)
         '/'            shift, and enter state 9
                        (reduce using rule 1)
         ')'            reduce using rule 1
         %eof           reduce using rule 1
-----------------------------------------------
Grammar Totals
-----------------------------------------------
Number of rules: 7
Number of terminals: 7
Number of non-terminals: 2
Number of states: 16
```

Abbildung 5.3: Happy-Infodatei für `Parser1.y` (leicht gekürzt)

Nach dem Erstellen kann man den Parser testen:

```
parser (lexer "1+2+3+4+5+6")
Plus (Plus (Plus (Plus (Plus (Number 1) (Number 2)) (Number 3)) (Number 4)) (Number 5)) (Number 6)
*Parser> parser (lexer "1/2+3-4+5*6")
Plus (Minus (Plus (Div (Number 1) (Number 2)) (Number 3)) (Number 4)) (Times (Number 5) (Number 6))
```

Generiert man den Parser mit der Option `-da` (`d` steht für „Debug“), so druckt der Parser beim Ausführen die Zustandsübergänge aus:

```
*Parser> parser (lexer "(4+1)*(4/5)-2")
state: 0,       token: 6,       action: shift, enter state 4
state: 4,       token: 1,       action: shift, enter state 3
state: 3,       token: 2,       action: reduce (rule 6), goto state 10
state: 10,      token: 2,       action: shift, enter state 6
state: 6,       token: 1,       action: shift, enter state 3
state: 3,       token: 7,       action: reduce (rule 6), goto state 15
state: 15,      token: 7,       action: reduce (rule 1), goto state 10
state: 10,      token: 7,       action: shift, enter state 11
state: 11,      token: 4,       action: reduce (rule 5), goto state 5
state: 5,       token: 4,       action: shift, enter state 8
state: 8,       token: 6,       action: shift, enter state 4
state: 4,       token: 1,       action: shift, enter state 3
state: 3,       token: 5,       action: reduce (rule 6), goto state 10
state: 10,      token: 5,       action: shift, enter state 9
state: 9,       token: 1,       action: shift, enter state 3
state: 3,       token: 7,       action: reduce (rule 6), goto state 12
state: 12,      token: 7,       action: reduce (rule 4), goto state 10
state: 10,      token: 7,       action: shift, enter state 11
state: 11,      token: 3,       action: reduce (rule 5), goto state 13
state: 13,      token: 3,       action: reduce (rule 3), goto state 5
state: 5,       token: 3,       action: shift, enter state 7
state: 7,       token: 1,       action: shift, enter state 3
state: 3,       token: 8,       action: reduce (rule 6), goto state 14
state: 14,      token: 8,       action: reduce (rule 2), goto state 5
state: 5,       token: 8,       action: accept.
Minus (Times (Plus (Number 4) (Number 1)) (Div (Number 4) (Number 5))) (Number 2)
```

```
{
module Parser where
import Data.Char

data Token = TokenInt Int
           | TokenSymbol Char
 deriving(Show)

lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
 | c `elem` ['+','-','*','/','(',')'] =
                 (TokenSymbol c) : lexer cs
 | isSpace c = lexer cs
 | isDigit c = lexNum (c:cs)
 | otherwise = error ("parse error, can't lex symbol " ++ show c)

lexNum cs = TokenInt (read num) : lexer rest
             where (num,rest) = span isDigit cs
}
%name parser
%tokentype { Token }
%token
        int        { TokenInt $$ }
        '+'        { TokenSymbol '+' }
        '-'        { TokenSymbol '-' }
        '*'        { TokenSymbol '*' }
        '/'        { TokenSymbol '/' }
        '('        { TokenSymbol '(' }
        ')'        { TokenSymbol ')' }
%left '+' '-'
%left '*' '/'
%%
E :: { Expr }
E : E '+' E         { Plus  $1 $3 }
  | E '-' E         { Minus $1 $3 }
  | E '*' E         { Times $1 $3 }
  | E '/' E         { Div $1 $3 }
  | '(' E ')'       { $2 }
  | int             { Number $1 }

{
happyError :: [Token] -> a
happyError [] = error "parse error: unerwartetes Ende"
happyError xs = error ("parse error:" ++ show xs)
data Expr   = Plus    Expr Expr
            | Minus   Expr Expr
            | Times   Expr Expr
            | Div     Expr Expr
            | Number  Int
  deriving(Show)
}
```

Abbildung 5.4: Happy-Parserspezifikation für arithmetische Ausdrücke (Parser2.hs)