

Entwicklung eines webbasierten EBNF-Simulators

Konstantin Klinger

Prüfer: Prof. Dr.-Ing. Dieter Pawelczak

Bachelorarbeit

eingereicht im Juni 2015

Erklärung

gemäß Beschluss des Prüfungsausschusses für die Fachhochschulstudiengänge der UniBwM vom 25.03.2010

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen.

Der Speicherung meiner Bachelor-/Masterarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

Neubiberg, den 25.06.2015

Konstantin Klinger

Inhaltsverzeichnis

Inhaltsverzeichnis.....	v
1 Einleitung.....	1
1.1 Motivation.....	1
1.2 Zielsetzung.....	1
1.3 Gliederung.....	1
2 Grundlagen.....	3
2.1 EBNF als formale Sprache.....	3
2.2 Lexikalische Analyse mit JFlex.....	4
2.3 Syntaktische Analyse mit CUP.....	5
2.4 Webanwendungen mit GWT.....	6
3 Entwicklung.....	9
3.1 Allgemeine Unterteilung.....	9
3.2 EBNF-Simulator.....	11
3.2.1 Systemübersicht.....	11
3.2.2 Scanner.....	13
3.2.3 Parser.....	15
3.2.4 Interpreter.....	23
3.3 GWT-Weboberfläche.....	37
3.3.1 Entwicklungszyklus.....	37
3.3.2 Integration der Java-Applikation.....	42
4 Evaluation.....	45
4.1 Test des EBNF-Simulators.....	45
4.1.1 Selbstdefinition der EBNF.....	45
4.1.2 Integer-Literale in Java.....	46
4.1.3 MiniB.....	47
4.2 Weboberfläche.....	48
5 Zusammenfassung.....	51
5.1 Fazit.....	51
5.2 Ausblick.....	51

5.3	Schlusswort	52
6	Anhang	53
6.1	Quellcode	53
6.1.1	lexer.jflex.....	53
6.1.2	parser.cup	55
6.1.3	Kommentare zu Klassendiagramm Interpreter.java.....	60
6.1.4	ListElement.java.....	61
6.1.5	Resources.java.....	62
6.2	Verwendete Software	63
6.3	Gliederung der beigelegten DVD	63
	Abbildungsverzeichnis	65
	Literaturverzeichnis.....	67

1 Einleitung

1.1 Motivation

„My duty as a teacher is to train, educate future programmers.“

- Niklaus Wirth [1]

Die Extended-Backus-Naur-Form (EBNF) gehört zu den formalen Sprachen. Sie beschreibt u.a. Grammatiken - den syntaktischen Aufbau - zahlreicher Programmiersprachen und bestimmt deren Attribute, Zeichen und Namensräume. In der Vorlesung „Programmerzeugungssysteme“ entstand durch intensive Beschäftigung mit der EBNF das Bedürfnis, selbstdefinierte Grammatiken auf syntaktische und dazu getätigte Eingaben auf semantische Korrektheit zu überprüfen, um die Lösung und Verbesserung von Übungsaufgaben, das Selbststudium und auch das Unterrichten zu vereinfachen. Ein derartiges Programm ist bisher nicht auf dem Markt zu finden und soll das Verständnis der Studenten, den zukünftigen Programmierern, über formale Sprachen, speziell der EBNF, steigern.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist somit die Entwicklung eines mit typischen Beispielen aus der Vorlesung verwendbaren EBNF-Simulators, der benutzerdefinierte EBNF-Grammatiken parst und anschließend diverse Eingaben zu dieser Syntax interpretiert. Dies soll innerhalb einer Webanwendung realisiert werden.

1.3 Gliederung

Im nachfolgenden Kapitel zwei [S. 3] werden zunächst die benötigten Grundlagen der EBNF, der lexikalischen und syntaktischen Analyse und des Entwickelns von Webanwendungen mit den verwendeten Generatoren bzw. Werkzeugsätzen beschrieben. Kapitel drei [S. 9] und vier [S. 45] enthalten die Entwicklung des Programms und dessen Evaluation. Abschließend wird in Kapitel fünf [S. 51] das Ergebnis resümiert.

2 Grundlagen

2.1 EBNF als formale Sprache

Softwareentwickler verfassen ihre Programme typischerweise in einer Programmiersprache. Diese werden von einem Compiler in maschinenverständliche Befehle, dem sog. Maschinencode, übersetzt, da Rechner nur fähig sind, derartige Befehlsfolgen zu interpretieren. Dieser komplexe Vorgang wird mit Hilfe von formalen Sprachen, die Programmiersprachen exakt beschreiben, systematisch strukturiert [2].

Ein erster Formalismus wurde 1960 von J. Backus und P. Naur durch die Backus-Naur-Form (BNF) eingeführt, deren Definition in [3] zu finden ist. Da in BNF-Grammatiken in der Regel rekursive Ausdrücke auftreten, wurde um 1977 von Herrn Niklaus Wirth die Extended-Backus-Naur-Form (EBNF) definiert. Mit dieser erweiterten BNF kann die Rekursion in der Beschreibung überwiegend vermieden werden. Die in Abbildung 2.1 zu sehende Selbstdefinition nach ISO-Standard (ISO/IEC 14977:1996 [4]) beschreibt die EBNF in der Wirth-Syntax-Notation [5, 6].

```
syntax = {production}.
production = identifier "=" expression ".".
expression = term { "|" term }.
term = factor {factor}.
factor = identifier | string | "(" expression ")" | "[" expression "]"
        | "{" expression "}".
identifier = letter { digit | letter }.
string = ""{ character }"".
letter = "a" | ... | "z" | "A" | ... | "Z".
digit = "0" | ... | "9".
```

Abbildung 2.1: Selbstdefinition der EBNF [7]

Die BNF wird damit hinsichtlich der Gruppierung mittels einfacher Klammern, Optionalität durch eckige Klammerung („*einmal oder keinmal*“ [8]) und Wiederholung dank geschweifeter Klammern („*keinmal oder beliebig oft*“ [8]) ergänzt. Eine Produktion wird durch ein Nichtterminal („*identifier*“) benannt und durch den Punkt als Endsymbol abgeschlossen. Sie stellt das Ergebnis einer syntaktischen Regel dar. Terminalsymbole („*string*“) stehen in Anführungsstrichen und lassen sich nicht weiter zerlegen. Für „*character*“ kann der gesamte ASCII-Zeichensatz [9] und ESC-Sequenzen [10] verwendet werden. Die Syntax einer in EBNF beschriebenen Sprache besteht somit aus beliebig vielen Produktionen. In Abbildung 2.2 findet sich ein zusammenfassender Überblick der EBNF-Kernelemente anhand eines Beispiels [11].

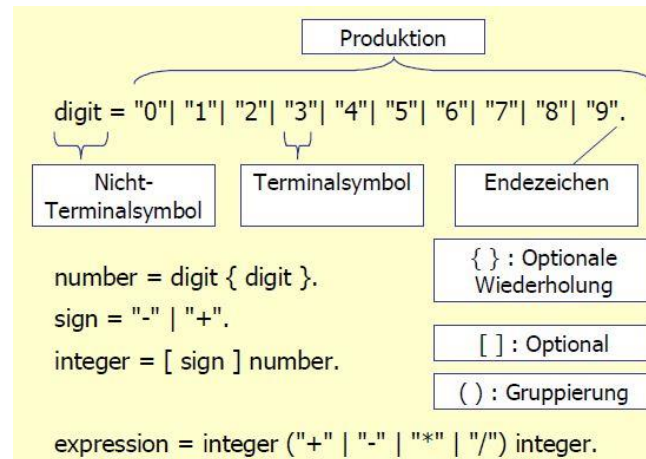


Abbildung 2.2: Kernelemente der EBNF [12]

Laut Wirth ist eine in EBNF beschriebene Sprache eine „Menge von Folgen von Terminalsymbolen“ [3], die durch „wiederholte Anwendung von syntaktischen Regeln“ [3] abgeleitet wird. Im Falle der EBNF werden also Nichtterminale ausgehend von einer Startregel Schritt für Schritt substituiert [3].

2.2 Lexikalische Analyse mit JFlex

Das Ziel der lexikalischen Analyse ist das Erkennen von Symbolen in einem bestimmten Quelltext und das Aufbereiten dieser für die syntaktische Analyse. Diese Aufgabe übernimmt ein Scanner, der den Quelltext einliest, überflüssige Zeichen, sog. „Whitespaces“¹, ignoriert, evtl. Kommentare entfernt und die Token aus der Quellsprache herausfiltert. Anschließend stellt er diese dem Parser beispielsweise in Form eines Symbolstroms zur Verfügung. Um die Programmierung einfacher zu gestalten und die Effizienz zu steigern, werden Lexer häufig durch sog. Scanner-Generatoren erzeugt [13].

In dieser Arbeit wird dazu JFlex („*The Fast Scanner Generator*“) verwendet, das als Java-Pendant zu *lex/flex* [14, 15] aus der Programmiersprache C gilt. Der 1998 von Gerwin Klein entwickelte, in Java geschriebene, quelloffene Generator erzeugt einen ebenfalls aus Java-Code bestehenden Scanner, der auf Deterministischen Endlichen Automaten oder DFAs („*deterministic finite automation*“ [16]) basiert. Dafür wird lediglich eine sog. JFlex-Datei benötigt, die in drei Abschnitte, getrennt durch „%%“, unterteilt ist [Abbildung 2.3].

Regeln zum Scannen des Quelltextes werden mit regulären Ausdrücken beschrieben. Bei mehr als einer zutreffenden wird die längste mögliche Zeichenfolge („*longest match*“ [17]) erkannt. JFlex überzeugt nicht nur durch eine schnelle, einfache Bedienung und Scannererzeugung, die

¹ Whitespaces (WS): Leerzeichen, Tabulatoren und Zeilenumbrüche

```
I.  User-Code

%%

II. Optionen und Deklarationen

%%

III.Regeln und Aktionen
```

Abbildung 2.3: Struktur der JFlex-Datei [19]

Unicode-Tauglichkeit oder die Möglichkeit, verschiedene Zustände („states“ [17]) zu verwalten, sondern vor allem auch durch die Zusammenarbeit u.a. mit dem Parser-Generator CUP, dessen Funktionalität im nachfolgenden Abschnitt erläutert wird [18, 19].

2.3 Syntaktische Analyse mit CUP

Nach der lexikalischen folgt die syntaktische Analyse, in der der eingegebene Quelltext hinsichtlich seiner grammatikalischen Korrektheit geprüft wird. Ein Parser analysiert die Grammatik der Eingabe, erzeugt bei Bedarf nützliche Fehlermeldungen und wandelt den Symbolstrom des Scanners in eine für die anschließende Semantikanalyse geeignete Form um. Häufig wird dabei die Darstellungsform eines syntaktischen Baumes gewählt. Semantische und syntaktische Analyse lassen sich jedoch nicht eindeutig voneinander trennen. Beispielsweise werden oftmals Klammern, die zu Gruppierungszwecken bezüglich der Auswertungsreihenfolge vorhanden sind, in Syntaxbäumen nicht mehr dargestellt, da der Parser diese bereits auswertet [20].

Prinzipiell lassen sich zwei verschiedene Parsevarianten unterscheiden. Beim Top-Down-Prinzip wird ein vorliegender Text ausgehend von einem Startsymbol (Hauptziel) geparkt. Angetroffene Nichtterminale werden dabei als Unterziele betrachtet. Der Syntaxbaum wird von oben nach unten aufgebaut.

Dieser wächst beim Bottom-Up-Prinzip hingegen von unten nach oben bis hin zur Wurzel. Hierbei wird ohne Vorgabe eines Ziels versucht, eingehende Symbole durch Nichtterminale zu ersetzen. In einem Schritt wird entweder ein weiteres Symbol gelesen („shift“) oder eine Symbolfolge wird einer Syntaxregel zugeordnet („reduce“) [21].

CUP („Construction of Useful Parsers“) ist eine 1996 an der Princeton University (USA) entwickelte und derzeit an der Technischen Universität München gepflegte, quelloffene Software, die in Java geschrieben ist und einen ebenfalls aus reinem Java-Code bestehenden Bottom-Up-Parser generiert. Dieser LALR(1)-Parser-Generator („Lookahead-LR“ [22]), der auch als Java-Pendant zu YACC/Bison [14, 23] aus der Programmiersprache C gilt, benötigt zur

Erzeugung des Parsers ausschließlich eine sog. CUP-Datei, in der die Symbole (Terminale und Nichtterminale), Produktionen (Grammatik-Regeln) und dazugehörige Aktionen definiert sind. Regeleingaben werden von CUP ausschließlich in BNF akzeptiert.

Dabei werden Prioritäten zum Lösen von typischen Shift-Reduce- bzw. Reduce-Reduce-Konflikten verwendet. Bei der einfachen arithmetischen Rechnung $2 \cdot 7 + 4$ könnte der Parser ohne die Regel „Punkt-Vor-Strich“, also Multiplikation mit höherer Wertigkeit als Addition, nicht entscheiden, welche Rechnung als erstes auszuführen ist. Lässt sich die vorhandene Symbolfolge also mehreren Syntaxregeln zuordnen, kann CUP ohne spezielle Richtlinie nicht beurteilen, welche Regel substituiert werden soll. Falls keine Prioritäten definiert sind, gilt die Konvention, dass die zuerst deklarierte Syntaxregel angewendet wird.

CUP beherrscht zudem die automatische Erzeugung und Ausgabe eines Abstrakten Syntaxbaumes oder AST („*abstract syntax tree*“ [24]). Neben einer teilweise automatisch erzeugten Fehlerausgabe ist zusätzlich eine Fehlerbehebung möglich [20, 25].

2.4 Webanwendungen mit GWT

Das Google Web Toolkit, oder kurz GWT (sprich engl. „*gwit*“), ist ein Werkzeugsatz zum Entwickeln und Optimieren von komplexen webbasierten Anwendungen, das seit dem 17.05.2006 als quelloffene Software jedem Anwender frei zur Verfügung steht. Ziel ist die Umsetzung des Java-Prinzips „*write once, run anywhere (WORA)*“ [26], speziell in Hinsicht auf die Browserunabhängigkeit von Webanwendungen. Eine in Java geschriebene GWT-Webseite ist auf allen modernen Browsern (inkl. mobiler Endgeräte) aufrufbar. Der Entwickler benötigt somit keine fundierten Kenntnisse über die verschiedenen Browsereigenschaften oder Entwicklungssprachen für webbasierte Anwendungen (z.B. JavaScript, XML, HTML, CSS).

Das Google Web Toolkit stellt also unter anderem die Funktionalität eines Compilers dar, das es den Java-Quellcode in ein lauffähiges, optimiertes JavaScript, HTML (Hypertext Markup Language) und CSS (Cascading Style Sheets) kompiliert. Der ursprünglich von Google entworfene Werkzeugsatz stellt außerdem eine umfassende Java-API-Bibliothek zum Entwerfen der grafischen Benutzeroberfläche (GUI) und das GWT-Developer-Plugin zum Debuggen in einer Java-Umgebung zur Verfügung. Des Weiteren werden auch direkte Eingaben in JavaScript, XML (Extensible Markup Language), HTML oder CSS parallel zu Java unterstützt. Allerdings werden in GWT nur die in [27] aufgelisteten Klassen und Methoden der JRE (Java Runtime Environment) unterstützt. Daher sind bei der Programmierung spezielle Lösungswege von Nöten, falls weitere Java-Funktionen benötigt werden.

Bei der Arbeit mit GWT trifft man häufig auf das Wort AJAX („*Kompositum aus Asynchronous JavaScript und XML*“ [28]). Dies ist ein Konzept der asynchronen, interaktiven Datenübertragung zwischen Server und Browser (Client), das bei modernen Webseiten (z.B.

Zoomen oder Verschieben der Karte bei GoogleMaps) unumgänglich ist, da bei einer Aktualisierung nicht die vollständige Seite, sondern jeweils nur einzelne sich ändernde Inhalte neu geladen werden. Während des Ladevorgangs ist die Webseite für den Nutzer nicht blockiert. Da das Google Web Toolkit ursprünglich als AJAX-Werkzeugsatz auf dem Markt war, soll dieser Name nun lediglich die Server-Client-Architektur verdeutlichen [29, 30].

3 Entwicklung

Nach dem Vorstellen der Grundlagen der EBNF und der verwendeten Software, wird nun die Entwicklung eines webbasierten EBNF-Simulators, im Folgenden „EBNF-Checker“, erläutert, dessen Logo in Abbildung 3.1 zu sehen ist.



Abbildung 3.1: Logo „EBNF-Checker“

3.1 Allgemeine Unterteilung

Abbildung 3.2 zeigt das Anwendungsfalldiagramm der gesamten Anwendung. Der Benutzer soll EBNF-Grammatiken eingeben und parsen können, wobei mit entsprechend sinnvollen Meldungen das Ergebnis der syntaktischen Analyse angezeigt wird. Als Zusatzfunktion ist das Speichern und Laden von persönlichen Grammatiken und das Anzeigen der Syntaxbäume der geparsen Regeln wünschenswert. Anschließend soll eine benutzerdefinierte Eingabe zu diesen Syntaxdefinitionen unter Angabe einer Startregel interpretiert werden können und alle Benutzereingaben sollten zurücksetzbar sein. Außerdem sind das Laden eines Beispiels und das Anzeigen einer Hilfsanleitung zur Benutzung nützlich.

Die Anwendung „EBNF-Checker“ lässt sich prinzipiell in zwei große Entwicklungsschritte unterteilen. In Anlehnung an das „Model-View-Controller“-Muster des Software Engineering Prozesses [31] erfolgt eine Aufteilung zwischen der Präsentation und der Programmsteuerung. Der nachfolgende Abschnitt befasst sich mit der Entwicklung eines Scanners und Parsers für EBNF-Grammatiken und anschließendem Interpretieren einer Eingabe zu diesen benutzerdefinierten Syntaxregeln. Die gesamte Funktionalität wird im Folgenden als „EBNF-Simulator“ bezeichnet und kann mit dem Modell des „Controllers“ verglichen werden.

Anschließend wird die Entstehung einer GWT-Weboberfläche und das Verbinden dieser mit dem Simulator beschrieben. Diese klassische grafische Benutzerschnittstelle stellt das Konzept der „View“ dar.

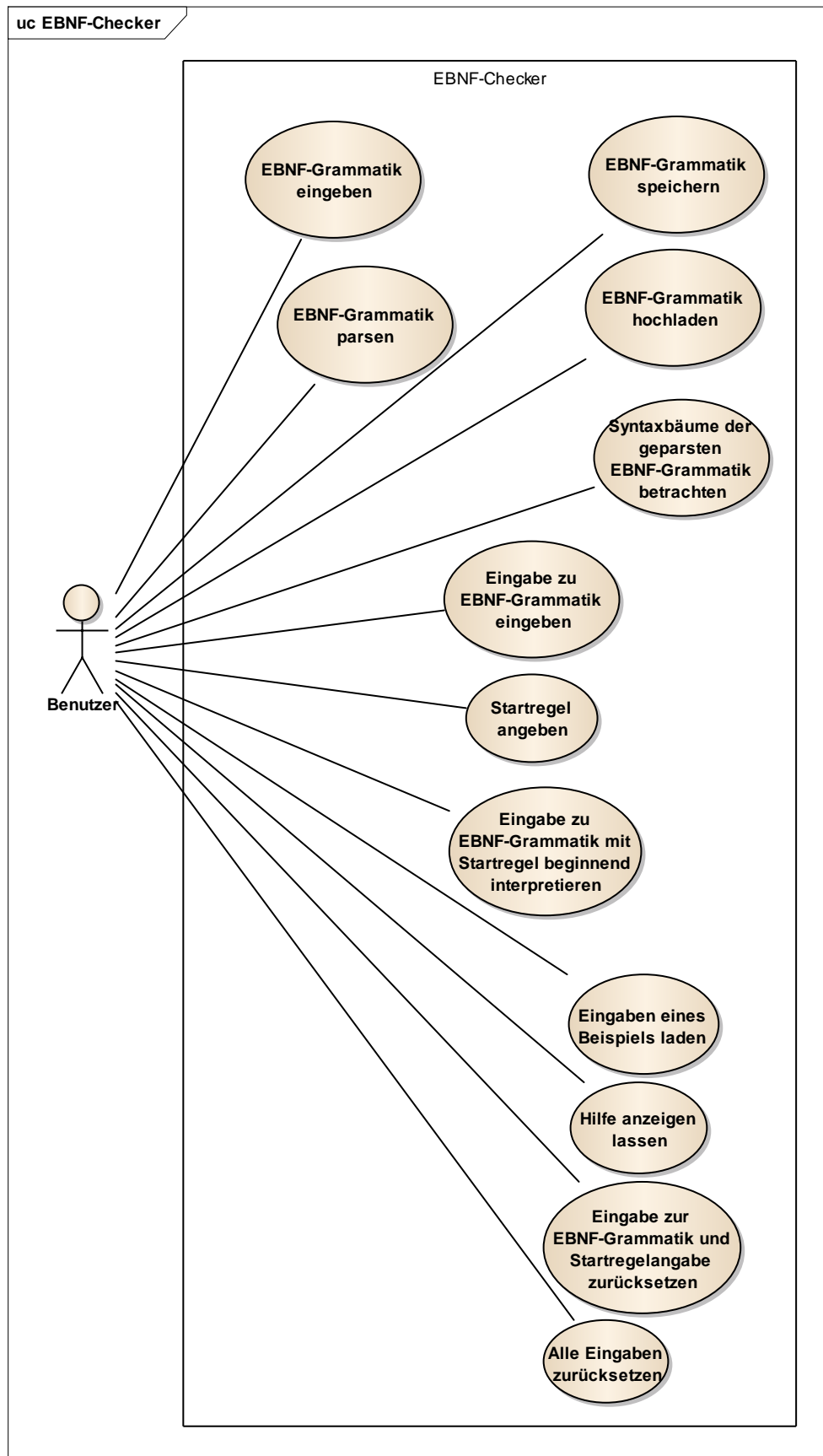


Abbildung 3.2: Übersicht über die Anwendungsfälle des Programms „EBNF-Checker“

3.2 EBNF-Simulator

3.2.1 Systemübersicht

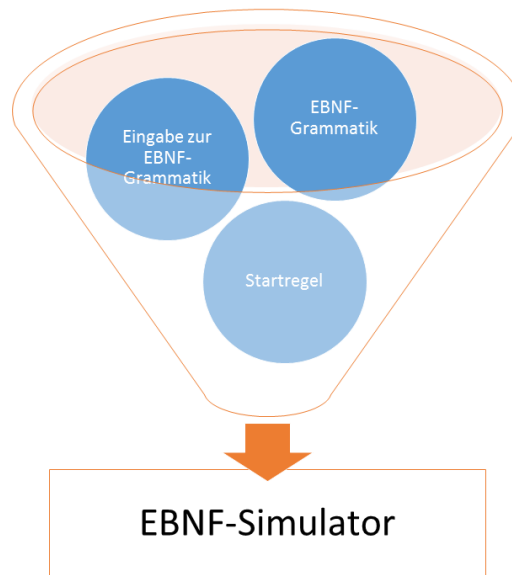


Abbildung 3.3: Notwendige Eingaben des Benutzers

Die Anwendung „EBNF-Simulator“ setzt sich aus drei Teilen zusammen und benötigt die in Abbildung 3.3 zu sehenden Benutzereingaben. Aus der Datei „*lexer.jflex*“ [Anhang 6.1.1] lässt sich der Java-Code des Scanners („*Scanner.java*“) mit Hilfe des Java-Archivs „*jflex-1.6.0.jar*“ erzeugen. Gleichermaßen entsteht aus „*parser.cup*“ [Anhang 6.1.2] durch „*java-cup-11b.jar*“ der Parser („*Parser.java*“) und die Tokendefinitionen („*ParserSym.java*“). Abbildung 3.4 und Abbildung 3.5 verdeutlichen das Vorgehen. Diese Klassen befinden sich im Package „*lexparse*“. Die für die Syntaxbaumerzeugung benötigten Dateien befinden sich im Package „*tree*“ und die des Interpreters in „*interpreter*“.

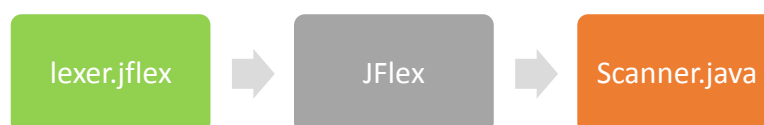


Abbildung 3.4: Erzeugen des Scanners mit JFlex

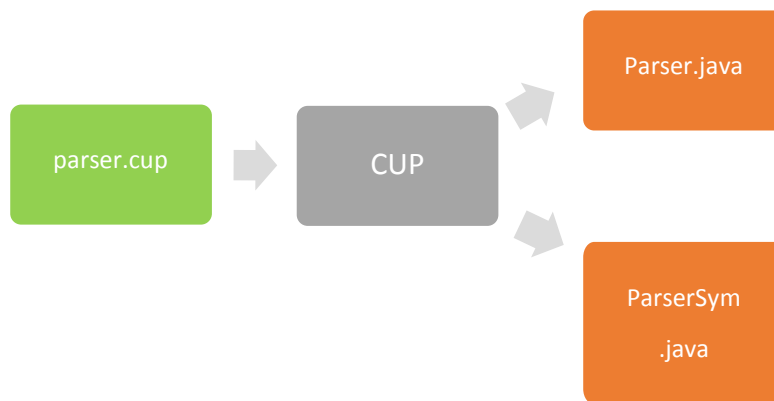


Abbildung 3.5: Erzeugen des Parsers mit CUP

Die Zusammenarbeit zwischen Scanner, Parser und Interpreter ist in Abbildung 3.6 dargestellt. Der Lexer scannt die eingegebene EBNF-Grammatik und leitet den erzeugten Symbolstrom an den Parser weiter. Dieser führt die syntaktische Analyse durch, informiert den Benutzer mit einer Ausgabe über den Verlauf des Scannens und Parsens und stellt dem Interpreter einen aus der Grammatik erzeugten Syntaxbaum zur Verfügung. Mit diesem und mit einer benutzerdefinierten Startregel wird eine zur EBNF-Grammatik getätigte Eingabe interpretiert. Das Ergebnis der semantischen Analyse wird wiederum in Form von sinnvollen Meldungen dargestellt.

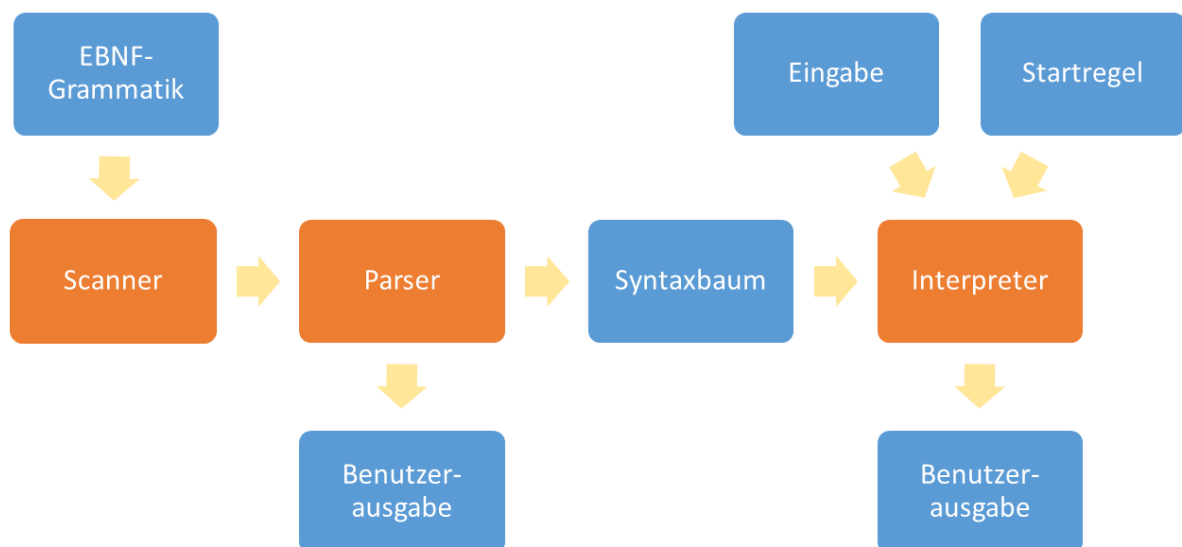


Abbildung 3.6: Systemübersicht „EBNF-Simulator“

3.2.2 Scanner

Zur Entwicklung des Scanners müssen zunächst die Symbole bzw. Token der EBNF definiert werden, die durch den generierten lexikalischen Analysator erkannt werden sollen. Dazu wird die Selbstdefinition der EBNF von Niklaus Wirth [Abbildung 2.1] bezüglich der Aufgabe und dem Zweck des zu entwickelnden Programms angepasst:

```
syntax = {production}.
production = nichtterminal "=" expression ".".
expression = term { "|" term }.
term = factor {factor}.
factor = nichtterminal | terminal | "(" expression ")"
        | "[" expression "]" | "{" expression "}".
nichtterminal = letter { digit | letter }.
terminal = ""{ character }"".
```

Abbildung 3.7: Angepasste Selbstdefinition der EBNF

Die Regeln „*identifizier*“ und „*string*“ werden in „*nichtterminal*“ und „*terminal*“ umbenannt, um den Begriffsbestimmungen aus den Vorlesungen „Programmerzeugungssysteme“ und „Grundlagen der Informatik“ näher zu kommen. Sie sind in der JFlex-Datei „*lexer.flex*“ [Anhang 6.1.1] wie folgt als lexikalische Muster beschrieben:

Tabelle 3.1: Lexikalische Muster der Regeln Terminal und Nichtterminal in *lexer.flex* [Anhang 6.1.1, Z. 88-92]

Name	EBNF	JFlex
TERMINAL	""{ character }""	"\"\"\" (([^\\"]+) \\\") \"\"\""
NICHTTERMINAL	letter { digit letter }	[a-zA-Z][a-zA-Z0-9]*

Die EBNF-Regel „*character*“ steht für den gesamten ASCII-Zeichensatz, sowie ESC-Sequenzen. Solch ein beliebiges Zeichen lässt sich in JFlex durch einen Punkt darstellen. Da ein Terminal jedoch von Hochkomma umschlossen ist, muss der Scanner diese auch als Trennzeichen erkennen. Deshalb wird das obige, etwas komplexe, Muster verwendet.

Des Weiteren werden in *lexer.flex* das Zuweisungs-, Oder- und Endsymbol und die Token für die drei verschiedenen Klammerarten (jeweils öffnend und schließend) definiert. Zeilenumbrüche, Tabulatoren und Leerzeichen, sog. „Whitespaces“, werden durch den Scanner ignoriert [Anhang 6.1.1, Z. 68-86]. Unerlaubte Zeichen führen durch die Hilfsfunktion „*error*“ zu einer entsprechenden Fehlermeldung [Anhang 6.1.1, Z. 45-52 & Z. 94-95]. Sie werden vom Parser aber ignoriert, sodass es zu keinem Programmabbruch kommt.

Im Konstruktor des Scanners wird festgelegt, dass die Grammatikeingabe von einem String gelesen und der erzeugte Symbolstrom durch eine sog. „*ComplexSymbolFactory*“ des CUP-Parsers verwaltet wird [Anhang 6.1.1, Z. 23-26]. Um im Parser die Position, Art und ggf. den Inhalt (bei Terminalen und Nichtterminalen) eines Symbols zeilen- und spaltengenau angeben zu können, wird die Hilfsfunktion „*symbol*“ benötigt. Diese fügt alle gelesenen Token in die „*ComplexSymbolFactory*“ hinzu [Anhang 6.1.1, Z. 28-43]. Auf eine separate Token-Klasse kann hier verzichtet werden, da diese von CUP erzeugt wird. Wichtig ist jedoch das „*%cup*“ in Zeile 16, um die Verbindung mit dem Parser-Generator zu aktivieren. Außerdem wird in der JFlex-Datei das Verhalten des Programms am Ende der Eingabe, beim Lesen des EOF-Symbols („*end of file*“), geregelt [Anhang 6.1.1, Z. 60-64].

Der Scanner analysiert die vom Benutzer eingegebene EBNF-Grammatik somit lexikalisch und bereitet diese auf die anschließende syntaktische Analyse vor.

3.2.3 Parser

a) Syntaktische Analyse

Die benutzerdefinierte Grammatik wird mit dem im Folgenden beschriebenen Parser hinsichtlich ihrer Syntax geprüft, nachdem der JFlex-Scanner sie lexikalisch untersucht hat. Unerlaubte Zeichen (lexikalische Fehler) werden, wie bereits beschrieben, ignoriert.

Zunächst wird die bereits veränderte EBNF-Selbstdefinition [Abbildung 3.7] erneut bearbeitet:

```
syntax = {production}.\nproduction = nichtterminal "=" alternative ".".\nalternative = reihe { "|" reihe }.\nreihe = factor {factor}.\nklammer = "(" alternative ")".\noption = "[" alternative "]".\nwiederholung = "{" alternative "}".\nfactor = nichtterminal | terminal | klammer | option | wiederholung.
```

Abbildung 3.8: Endgültig angepasste Selbstdefinition der EBNF

Die Regeln „*expression*“ und „*term*“ wurden in „*alternative*“ und „*reihe*“ umbenannt, um die Prinzipien der Reihen- und Oderverschachtelung der EBNF zu verdeutlichen. Da die Hauptzielgruppe dieses Programms vorerst Studenten sind, die ihre ersten Erfahrungen mit formalen Sprachen sammeln, soll so das Verständnis erleichtert werden. Deshalb wurden auch die EBNF-Elemente der Gruppierung („*klammer*“), Optionalität („*option*“) und der Wiederholung („*wiederholung*“) als separate Regeln definiert. Die Definitionen der Nichtterminalsymbole „*terminal*“ und „*nichtterminal*“ haben bezüglich Tabelle 3.1 keinerlei Änderungen mehr erfahren. Ein weiterer Vorteil dieser Umwandlung ist die im Interpreter mögliche getrennte Behandlung der einzelnen EBNF-Kernfunktionen², da diese nun jeweils in eigenen Regeln vorliegen.

Die Token des Scanners³, die, wie in Abschnitt 3.2.2 bereits beschrieben, lediglich in der CUP-Datei „*parser.cup*“ [Anhang 6.1.2] zu definieren sind, werden in den Zeilen 79-82 als sog. Terminalsymbole des Parsers, typischerweise in Großbuchstaben, deklariert. Die Zeichen „NIGHTTERMINAL“ und „TERMINAL“ übergeben ihren Inhalt in Form eines Strings. Die restlichen Symbole benötigen keinen Datentyp, da sie keine Werte transportieren. Jedoch

² EBNF-Kernfunktionen: Gruppierung („()“), Optionalität („[]“), Wiederholung („{ }“), Reihen- und Oderverschachtelung von Terminal- und Nichtterminalsymbolen

³ Token: NIGHTTERMINAL, TERMINAL, ZUWEISUNG, ENDE, ODER, KLAMMER_AUF, KLAMMER_ZU, OPTION_AUF, OPTION_ZU, WDH_AUF, WDH_ZU

erhalten die Klammersymbole entsprechend ihrer Wertigkeit verschiedene Prioritäten [Abbildung 3.9], die in Abbildung 3.10 dargestellt sind. Damit ist beim Parsen eine eindeutige Rangfolge festgelegt.

```
//Prioritäten
precedence left OPTION_AUF, OPTION_ZU;
precedence left WDH_AUF, WDH_ZU;
precedence left KLAMMER_AUF, KLAMMER_ZU;
```

Abbildung 3.9: Klammerprioritäten in *parser.cup* [Anhang 6.1.2, Z. 95-98]

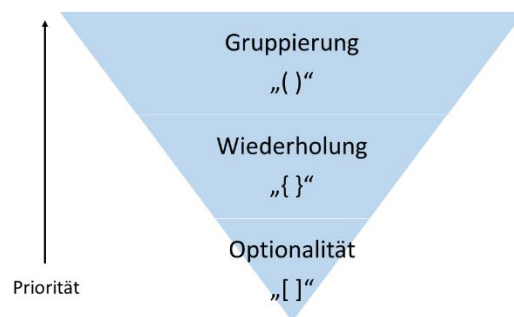


Abbildung 3.10: Prioritäten der EBNF-Klammerarten

Da die in Kapitel 2.3 beschriebene CUP-Spezifikation Grammatikregeln für den generierenden Parser ausschließlich in BNF erlaubt, muss die an die Gegebenheiten dieses Programms angepasste EBNF-Selbstdefinition [Abbildung 3.8] in die einfache Backus-Naur-Form umgewandelt werden [Abbildung 3.11]. In dieser Gestalt liegt die Syntax in „*parser.cup*“ vor [Anhang 6.1.2, Z. 111ff.]. Die einzelnen Regeln des Parsers sind zuvor als sog. Nichtterminalsymbole deklariert [Anhang 6.1.2, Z. 84-93]. Die zusätzliche Grammatikregel „*initial*“ in den Zeilen 101-109 wird benötigt, um die im nachfolgenden Abschnitt beschriebenen Syntaxbäume in einer einzigen HashMap („*treeMap*“) einzuordnen.

```
syntax ::= syntax production | syntax;
production ::= NICHTTERMINAL ZUWEISUNG alternative ENDE;
alternative ::= alternative ODER reihe | reihe;
reihe ::= reihe factor | factor;
klammer ::= KLAMMER_AUF alternative KLAMMER_ZU;
option ::= OPTION_AUF alternative OPTION_ZU;
wiederholung ::= WDH_AUF alternative WDH_ZU;
factor ::= NICHTTERMINAL | TERMINAL | klammer | option | wiederholung;
```

Abbildung 3.11: Angepasste EBNF-Selbstdefinition [Abbildung 3.8] in BNF zur Verwendung in Java-CUP

b) Syntaxbaumerzeugung

Um im nächsten Schritt der Semantikanalyse eine bestimmte Eingabe des Benutzers zu einer korrekt gescannten und geparsten EBNF-Grammatik auf Korrektheit überprüfen zu können, benötigt der Interpreter eine möglichst einfache und für den Computer verständliche Repräsentation der eingegebenen Regeln. Hierfür wird in der Informatik der Syntaxbaum verwendet, der die Syntax hierarchisch in einer Baumstruktur anordnet.

Im Folgenden wird der Begriff Parse-Baum [32] für die grafische Repräsentation einer Regel als Ergebnis der syntaktischen Analyse verwendet. In dieser Form wird die Baumstruktur während des Parsens erzeugt. Hingegen wird als Eingabe des Interpreters von einem optimierten Syntaxbaum gesprochen, da der Parse-Baum vor der semantischen Analyse vereinfacht wird.

In diesem Programm wird pro eingegebener EBNF-Regel zunächst ein binärer Parse-Baum erstellt. Die von CUP angebotene automatische AST-Erzeugung wird aufgrund ihrer umständlichen Umsetzung und des erschwerten Zugriffs auf die Elemente des Abstrakten Syntaxbaumes nicht verwendet.

Der Parse-Baum wird mit den Java-Klassen „*BinaryTree*“, „*Node*“ und „*Type*“ aus dem Package „*tree*“ manuell in den Aktionen der BNF-Grammatikregeln (eingeschlossen von „*{:...:}*“) der CUP-Datei „*parser.cup*“ erzeugt [Anhang 6.1.2, Z. 111ff.]. Ein Knoten (engl. „*node*“) erhält dort zunächst fünf Attribute, die in den folgenden vereinfachten Klassendiagrammen dargestellt sind.

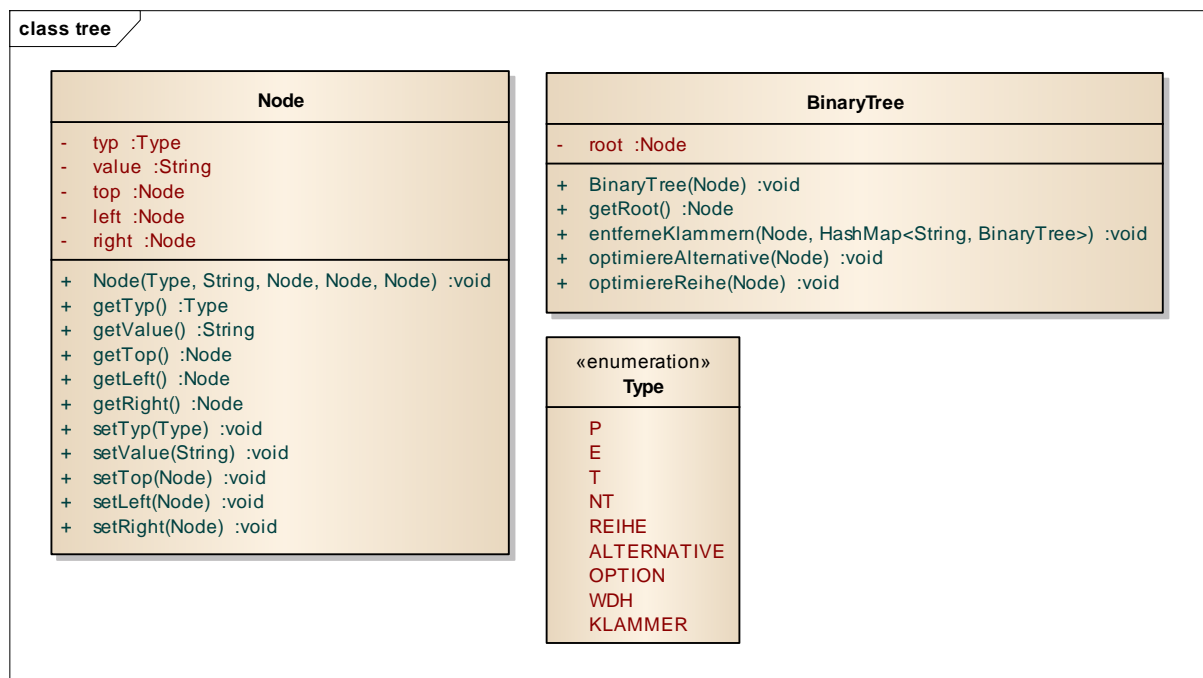


Abbildung 3.12: Klassendiagramme der Klassen „*Node*“, „*Type*“ und „*BinaryTree*“ (Syntaxbaumerzeugung)

Ausgehend von der Wurzel des Baumes, welche immer vom Typ „*P*“ (Produktion) ist, wird als rechter Nachfolger ein Knoten vom Typ „*E*“ (Ende), der symbolisch für den Punkt als

Endzeichen einer jeden EBNF-Regel steht, und als linker Nachfolger der Teilbaum mit den entsprechenden Regeln dieser Produktion eingefügt. Dieses Prinzip ist in Abbildung 3.13 dargestellt.

Links von Reihen und Alternativen steht deren erstes beinhaltende Element der logischen Verknüpfung: Terminal („*T*“), Nichtterminal („*NT*“), Option („*OPTION*“), Wiederholung („*WDH*“) oder erneut eine Reihe bzw. Alternative. Rechts von ihnen werden hingegen die nächsten Reihen oder Alternativen eingefügt, mit Ausnahme des letzten Knotens.

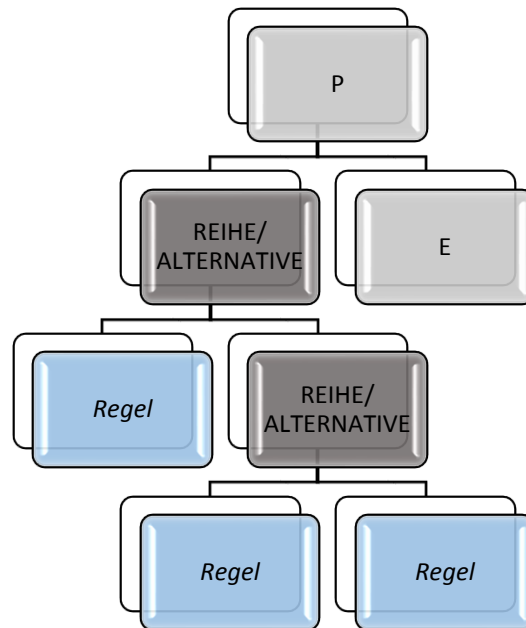


Abbildung 3.13: Prinzipieller Aufbau der Syntaxbäume

Wie in Abbildung 3.14 zu sehen ist, besitzen Optionen und Wiederholungen keinen rechten Nachfolger, lediglich links wird der Inhalt der entsprechenden Regel eingefügt. In Abbildung 3.15 ist am Beispiel der Regel „wiederholung“ die Umsetzung dieses Prinzips der Baumerzeugung in den Aktionen der Grammatikregeln der CUP-Datei dargestellt. Es wird ein neuer Knoten vom Typ „*WDH*“ als Ergebnis zurückgegeben, der auf den Inhalt der Wiederholungsregel als linken Nachfolger verweist.

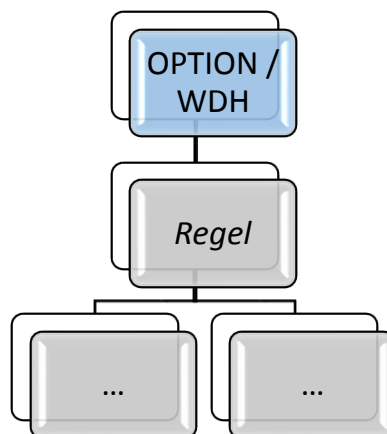


Abbildung 3.14: Wiederholungen und Optionen im Syntaxbaum


```
wiederholung ::= WDH_AUF:w alternative:a WDH_ZU:z
{
  Node temp = new Node(Type.WDH, null, null, a, null);
  a.setTop(temp);
  RESULT = temp;
:}
;
```

Abbildung 3.15: Syntaxbaumerzeugung der Regel „wiederholung“ in *parser.cup* [Anhang 6.1.2, Z. 232 - 239]

Zur Verdeutlichung des Prinzips der Baumerzeugung soll das folgende Beispiel dienen:

```
digit = "0" | "1" | "..." | "9".
number = digit { digit }.
float = number "." number.
```

Abbildung 3.16: Grammatikbeispiel zur Syntaxbaumerzeugung

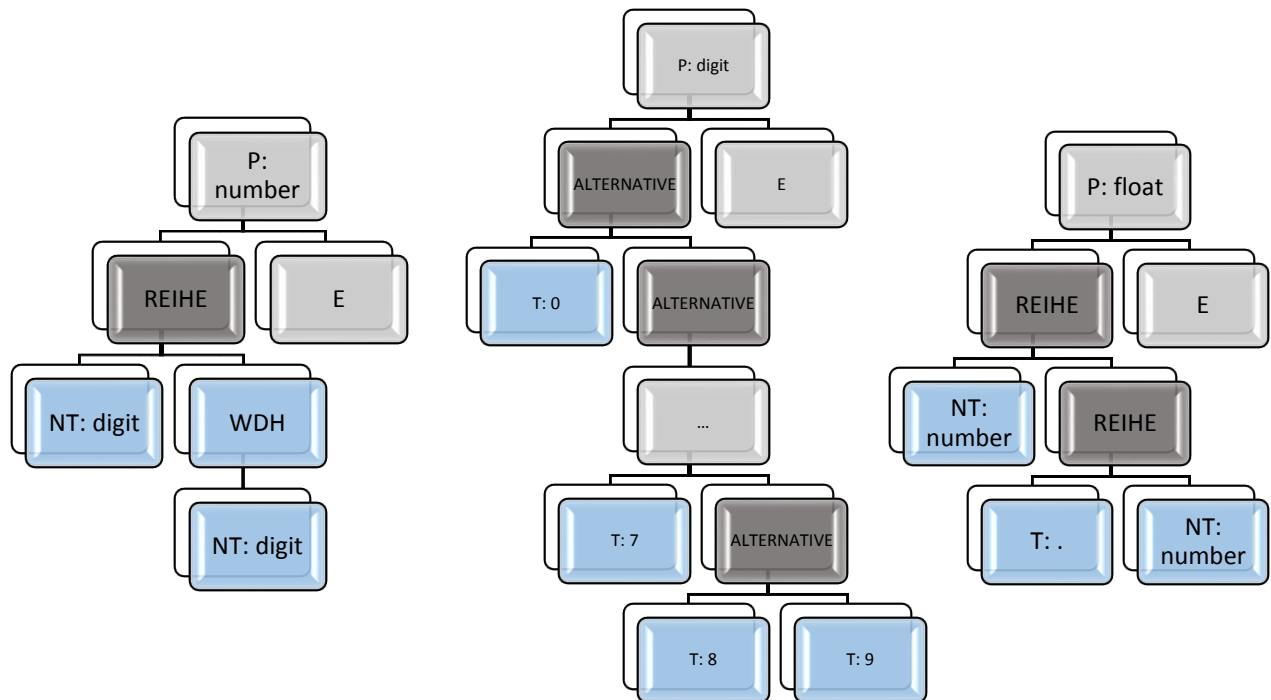


Abbildung 3.17: Optimierte Syntaxbäume des Grammatikbeispiels [Abbildung 3.16]

Diese drei EBNF-Definitionen erzeugen die in Abbildung 3.17 zu sehenden optimierten Syntaxbäume.

Die Herausforderung beim Programmieren der Aktionen zur Erstellung der Syntaxbäume betrifft unter anderem die Bottom-Up-Architektur des CUP-Parsers. Der Baum wird von unten nach oben bzw. vom Blatt zur Wurzel zusammengestellt. Das ist einer der Gründe, weshalb vor dem Interpretieren noch einige Optimierungen am Parse-Baum vorgenommen werden müssen. Danach ist die Basis für die anschließende semantische Analyse geschaffen.

Reihen und Alternativen können in der hier verwendeten angepassten EBNF-Selbstdefinition [Abbildung 3.8] mit der Startregel „*syntax*“ niemals alleine aufgerufen werden. Bei jedem Parsevorgang einer Alternative-Regel wird anschließend auch die Reihe-Regel durchlaufen. Jedoch kann aufgrund der rekursiven Verschachtelung der EBNF-Regeln auch keine Reihe erzeugt werden, ohne durch die Alternative-Regel zu laufen. Deshalb werden überflüssige Knoten aus dem Parse-Baum entfernt. Abbildung 3.18 zeigt die Darstellung der Regel „*digit*“ aus Abbildung 3.16 vor dieser Optimierung.

Des Weiteren gibt es im optimierten Syntaxbaum keine Klammer-Regel. Sämtliche in der eingegebenen EBNF-Grammatik und im Parse-Baum des Parsers auftretenden Klammern, die ohnehin nur zur Gruppierung verwendet werden, werden als separate Syntaxbäume mit fortlaufender Nummerierung, beginnend mit „*klammer0*“, dargestellt. Dem Interpreter wird hiermit die Auswertung einer zusätzlichen Regel erspart.

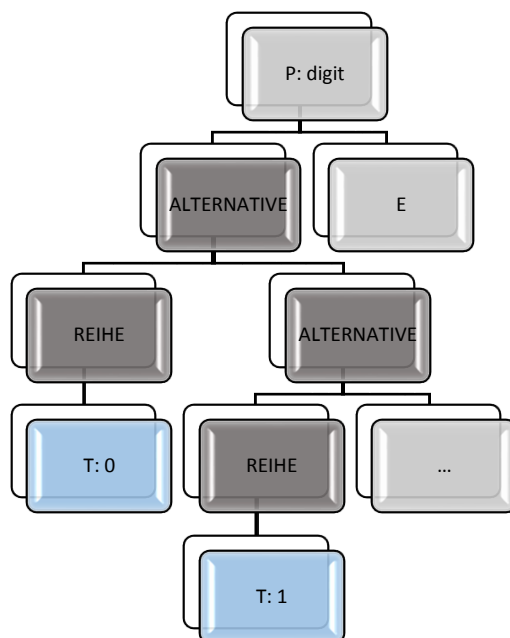


Abbildung 3.18: Parse-Baum der Regel „*digit*“ [Abbildung 3.16] vor Optimierung

Somit ist die durch den Benutzer definierte EBNF-Syntax gescannt, geparkt und in Form eines optimierten Syntaxbaumes auf das Interpretieren einer Benutzereingabe vorbereitet.

c) Fehlerbehandlung

Ein weiterer Aspekt, der für die lehrende Benutzung dieser Software eine große Bedeutung hat, ist das Verhalten des Programms bei Syntaxfehlern. Falls es während des Parsens zu syntaktischen Unstimmigkeiten der eingegebenen Grammatik in Bezug auf die EBNF [Abbildung 3.8] kommen sollte, werden passende Fehlermeldungen generiert. Dazu werden die bereits in CUP implementierten Funktionen „*report_error*“, „*report_fatal_error*“ und „*syntax_error*“ in „*parser.cup*“ überschrieben [Anhang 6.1.2, Z. 25-44], wobei die im Scanner erfolgte Abspeicherung der Position der Token genutzt wird [Anhang 6.1.1, Z. 28-43].

Der „EBNF-Simulator“ erzeugt für das in Abbildung 3.19 dargestellte Beispiel einer fehlerhaften Grammatikeingabe vier verschiedene Fehlermeldungen [Abbildung 3.20].

```
boolean = "0" | "1".  
boolean = false | true.  
1fault = "FEHLER"=  
2fault = "FEHLER"..
```

Abbildung 3.19: Beispiel für fehlerhafte Grammatikeingabe

Zunächst wird vom Scanner das unerlaubte Symbol „|“ gemeldet, da Nichtterminale laut Definition [Abbildung 3.7] nicht mit Ziffern beginnen dürfen. Dabei handelt es sich um einen lexikalischen Fehler.

Anschließend wird auf das doppelte Auftreten der Regel „*boolean*“ hingewiesen, wobei nur für die zuerst deklarierte ein Syntaxbaum erzeugt wird. Dieser Schritt der semantischen Analyse wird dem Interpreter vorweggenommen und bereits im Parser behandelt.

Trotz des lexikalischen (Zeichen ignoriert) und des semantischen Fehlers (nur ein Parse-Baum erzeugt) kann die Syntaxanalyse zunächst fortgesetzt werden. Der Abbruch erfolgt erst beim Auftreten des Symbols „=“ vor Ende der Definition „*1fault*“, da CUP nach diesem syntaktischen Fehler nicht weiterparsen kann. Das ist auch der Grund, weshalb sich keine Meldung zur letzten fehlerbehafteten Regel „*2fault*“ finden lässt, in der ein unerlaubtes Symbol („2“) und ein Syntaxfehler („..“) auftritt.

```
Error at line 3, column 1: Illegal character "1" (ignored)  
Error for input symbol NICHTTERMINAL "boolean" in line 2, column 7: This  
rule is already defined  
Error for input symbol ZUWEISUNG "=" in line 3, column 18: Syntax error  
Error for input symbol ZUWEISUNG "=" in line 3, column 18: Couldn't repair  
and continue parse
```

Abbildung 3.20: Erzeugte Fehlermeldungen bei Eingabe des fehlerhaften Grammatikbeispiels [Abbildung 3.19]

Neben dem in diesem Beispiel auftretenden semantischen Fehler einer mehrfachen Regeldefinition werden im Parser außerdem linkseitig-rekursive Produktionen herausgefiltert, da derartige Konstrukte bisher nicht durch den Interpreter unterstützt werden. Der Benutzer erhält eine entsprechende Meldung [Abbildung 3.21] und kann ohne Korrektur der Links-Rekursion keine Eingaben zur Grammatik interpretieren.

```
Error for input symbol NICHTTERMINAL "left" in line 1, column 4: This rule
is left-recursive (interpreting not supported)
```

Abbildung 3.21: Fehlermeldung bei linksrekursiven Grammatikeingaben

Bei erfolgreich geparsten EBNF-Grammatiken ohne Fehlermeldung erhält der Benutzer die in Abbildung 3.22 zu sehende Rückmeldung des „EBNF-Simulators“.

```
Result: CORRECT (parsed without errors)
```

Abbildung 3.22: Erfolgreiches Parsen ohne Fehler

3.2.4 Interpreter

Nachdem die vom Benutzer definierte EBNF-Grammatik lexikalisch und syntaktisch korrekt analysiert und optimierte Syntaxbäume als Schnittstelle zum Interpreter erzeugt werden, übernimmt dieser die semantische Analyse einer dazu getätigten Eingabe. In diesem Abschnitt wird die komplexe Funktionalität des Interpreters vereinfacht und abstrakt erläutert. Die dargestellten UML-Diagramme spiegeln nicht exakt die programmiertechnische Umsetzung wieder.

Die größte Herausforderung betrifft im Wesentlichen die Entwicklung eines Konzepts, mit dem die optimierten Syntaxbäume zum Prüfen einer Eingabe durchlaufen werden. Dabei kann nicht mehr auf die Hilfe eines Generators zurückgegriffen werden. Den prinzipiellen Aufbau der Klasse „*Interpreter*“ aus dem Package „*interpreter*“ zeigt das Klassendiagramm in Abbildung 3.23. Es wird jedoch ausdrücklich darauf hingewiesen, dass es sich hierbei um keine vollständige Darstellung handelt, sondern ausschließlich um eine Auflistung der zum Verständnis notwendigen Attribute und Methoden, die in Anhang 6.1.3 genauer beschrieben sind.

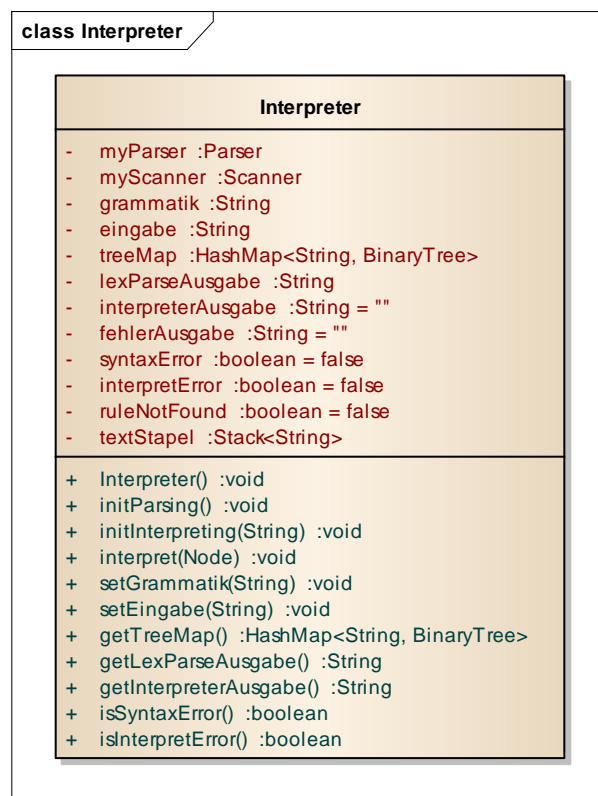


Abbildung 3.23: Klassendiagramm „*Interpreter*“

Ein vom Parser übergebener optimierter Syntaxbaum wird beim Interpretieren von oben nach unten (Top-Down) durchlaufen. Dies hat im Vergleich zur Bottom-Up-Variante den Vorteil, dass keine komplexen Shift- und Reduce-Operationen durchgeführt werden müssen. Da die Eingabe zur EBNF-Grammatik nicht gescannt wird und somit nicht in Form eines

Symbolstroms vorliegt, sondern gebündelt in einem String übergeben wird, können die einzelnen Token jedoch zu Beginn nicht voneinander separiert werden. Außerdem bedarf es bei dieser Art der Realisierung einen speziellen Umgang mit linksrekursiven Grammatikregeln, die ansonsten zu Problemen führen könnten.

In den folgenden Abschnitten wird die Umsetzung der Idee beschrieben, die Bäume mit der rekursiven Methode „*interpret*“, bei einer angegebenen Startregel beginnend, abwärts zu durchlaufen. Dabei wird eine Typunterscheidung zwischen den Regeln Terminal, Nichtterminal, Reihe, Alternative, Option und Wiederholung mit einem Switch-Case-Konstrukt [Abbildung 3.24] vorgenommen und die Funktion „*interpret*“ wiederholt rekursiv aufgerufen. Diese gibt für eine korrekte Überprüfung den Integer-Wert „1“ und für eine fehlerhafte „-1“ zurück, was in den nachfolgenden Aktivitätsdiagrammen als „Richtig“ und „Falsch“ bezeichnet wird. In den jeweiligen Fällen erfolgt die semantische Überprüfung der Eingabe mit der entsprechenden Syntaxregel. Für die Klammer zur Gruppierung („*KLAMMER*“) werden bereits im Parser [Kapitel 3.2.3b)] separate Syntaxbäume erzeugt, die dann vom Interpreter wie gewöhnliche Produktionen ausgewertet werden.

```
public int interpret(Node regel) {
    switch(regel.getTyp()) {
        case T:
            /*Behandlung der Regel Terminal*/
        case NT:
            /*Behandlung der Regel Nichtterminal*/
        case REIHE:
            /*Behandlung der Regel Reihe*/
        case ALTERNATIVE:
            /*Behandlung der Regel Alternative*/
        case OPTION:
            /*Behandlung der Regel Option*/
        case WDH:
            /*Behandlung der Regel Wiederholung*/
        default:
            return -1;
    }
}
```

Abbildung 3.24: Typunterscheidung mit Switch-Case-Konstrukt

a) Terminal

Ein Terminal ist die Basis der Syntax. Alle Regeln lassen sich durch Substitutionen in eine Folge von Terminalsymbolen umwandeln. Die einzelnen Zeichen der zu interpretierenden Eingabe des Benutzers liegen im Stack „*textStapel*“ und der Inhalt, der in diesem Schritt zu prüfenden Terminalregel, im String „*regel.getValue*“. Deshalb werden die Symbole zeichenweise verglichen, indem mit einer For-Schleife über die Anzahl der Zeichen des Terminals navigiert wird. Im Fehlerfall müssen zuvor aus dem „*textStapel*“ entnommene Symbole wieder zurückgelegt werden und der Vorgang gibt ein negatives Ergebnis zurück. Bei korrekter Überprüfung wird die Schleife bis zum Ende durchlaufen. In Abbildung 3.25 ist dieser Ablauf mit einem Aktivitätsdiagramm visualisiert.

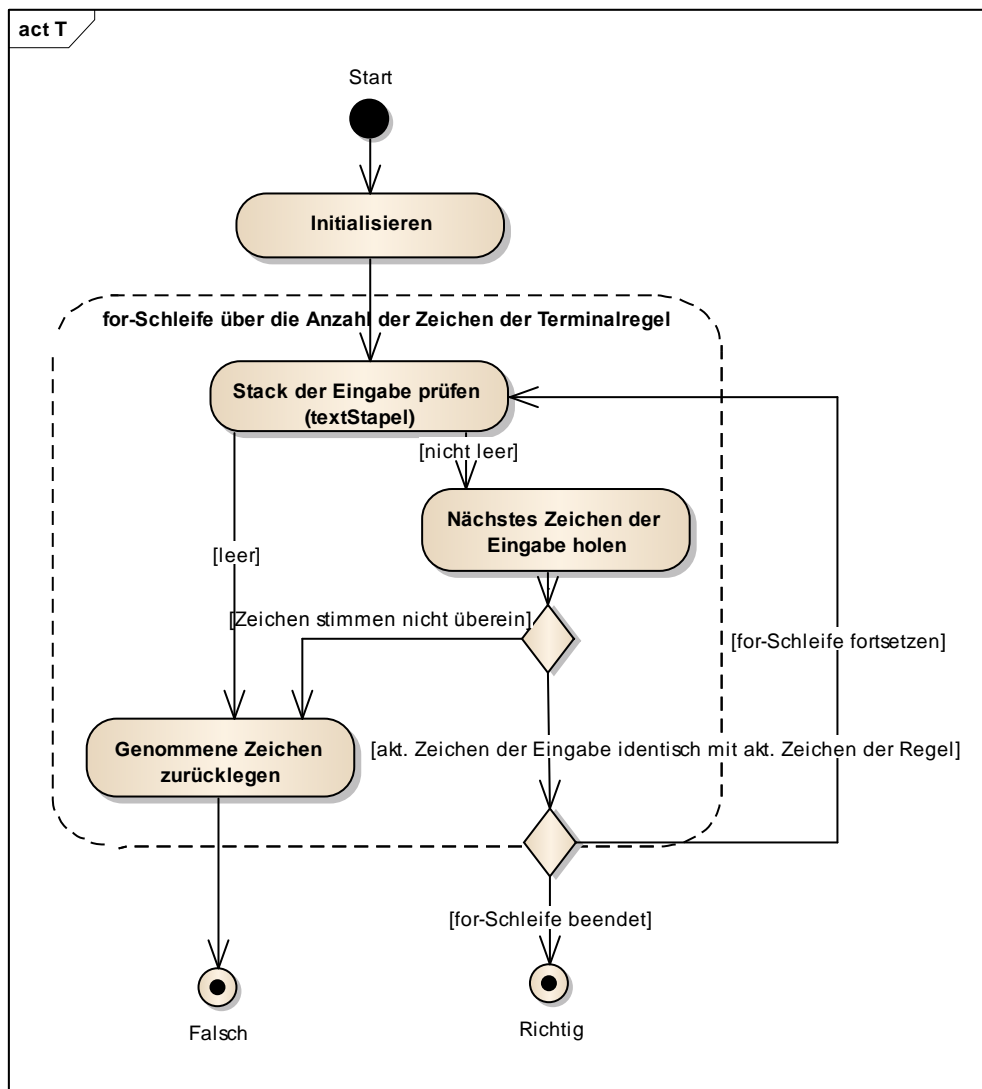


Abbildung 3.25: Aktivitätsdiagramm der Regel Terminal („T“)

Beim Vergleichen der Eingabe „AB“ mit der Terminalregel „ABC“ werden die ersten beiden Zeichen richtig erkannt. Jedoch ist der Eingabestapel beim dritten Schleifendurchlauf leer, weshalb die genommenen Symbole „A“ und „B“ wieder auf dem Stack („*textStapel*“) landen und als Resultat der Überprüfung „Falsch“ zurückgegeben wird.

b) Nichtterminal

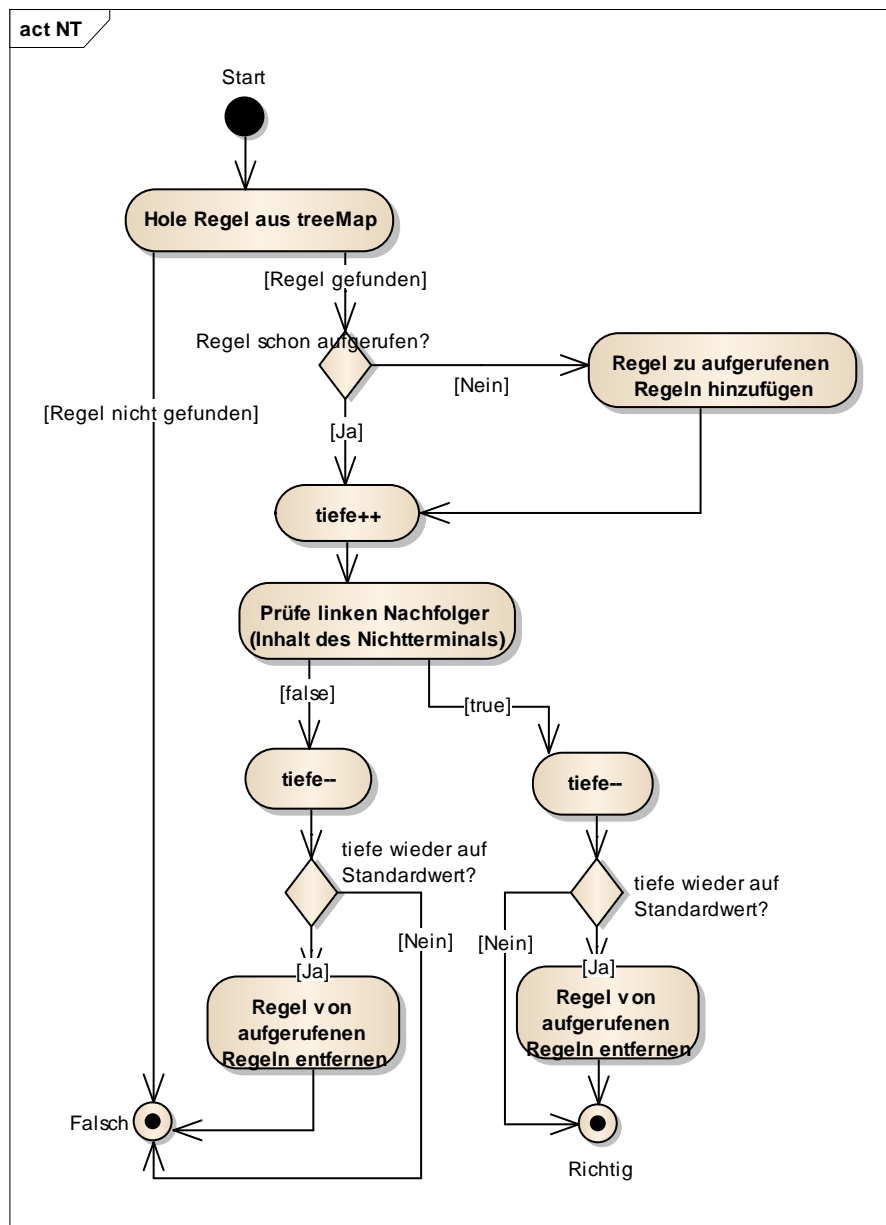


Abbildung 3.26: Aktivitätsdiagramm der Regel Nichtterminal („NT“)

Da das Problem von mehrfach definierten Syntaxregeln bereits im Parser behandelt wird [Kapitel 3.2.3c)], erfolgt bei der Auswertung eines Nichtterminals zunächst die Kontrolle, ob überhaupt eine derart benannte Produktion in die HashMap der Syntaxbäume („treeMap“) hinzugefügt wurde. In Abbildung 3.26 ist sichtbar, dass bei gefundener Definition zunächst die Tiefenvariable erhöht und ggf. die Liste aller aufgerufenen Regeln ergänzt wird. Nach anschließendem rekursivem Aufruf der „interpret“-Funktion und Interpretation des Inhalts der Nichtterminalregel („regel.getLeft“) werden diese beiden Schritte wieder umgekehrt. Der Zweck der Speicherung der Tiefe und der durchlaufenen Produktionen ist die Lösung der Rekursion, die in Abschnitt g) erklärt wird.

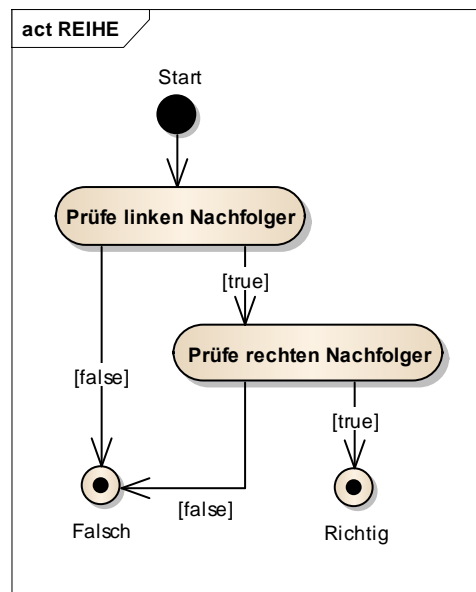
c) Reihe

Abbildung 3.27: Aktivitätsdiagramm der Regel Reihe („REIHE“)

Die Aneinanderreihung mehrerer Syntaxregeln stellt eine logische UND-Verknüpfung dar. Falls die Überprüfung des linken Nachfolgers scheitert, kann der Vorgang sofort mit „Falsch“ abgebrochen werden. Nur wenn linker und rechter anhängender Teilbaum fehlerfrei interpretiert werden, kann der Rückgabewert „Richtig“ ergeben [Abbildung 3.27].

d) Alternative

Vergleichbar zur Reihe besteht auch die Alternative mit ODER-Verknüpfungen aus einer einfachen Logik. Sollte bei der Kontrolle der zuerst interpretierten Wahlmöglichkeit bereits eine Übereinstimmung mit der Eingabe entdeckt werden, bedarf es keinerlei Inspektion der zweiten, da nur eine der beiden Alternativmöglichkeiten für eine korrekte Interpretation erfüllt sein muss.

Allerdings haben Tests der Anwendung ergeben, dass in manchen Fällen erst nach Vertauschen aller bzw. einzelner Alternativen das erwartete Ergebnis produziert wird. Deshalb entsteht das geringfügig komplexere Aktivitätsdiagramm aus Abbildung 3.28. Nach erfolglosem Interpretieren einer Eingabe zu einer vom Benutzer angegebenen EBNF-Syntax wird ein sog. „zweiter Versuch“ mit umgekehrten Alternativen gestartet.

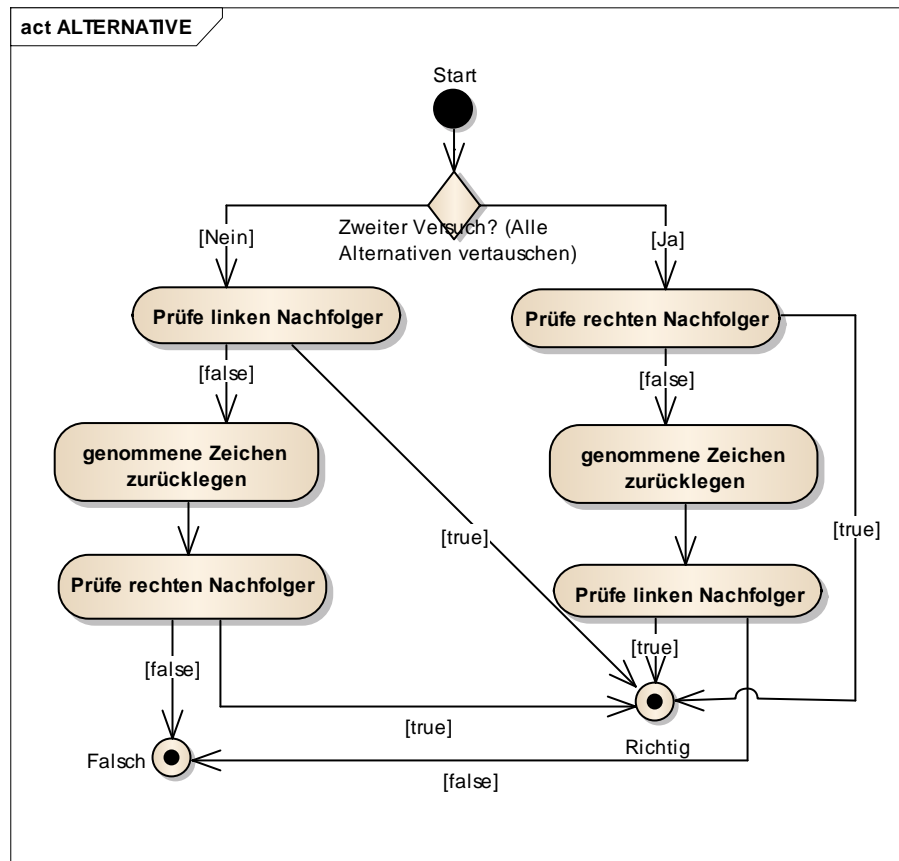


Abbildung 3.28: Aktivitätsdiagramm der Regel Alternative („ALTERNATIVE“)

e) Option

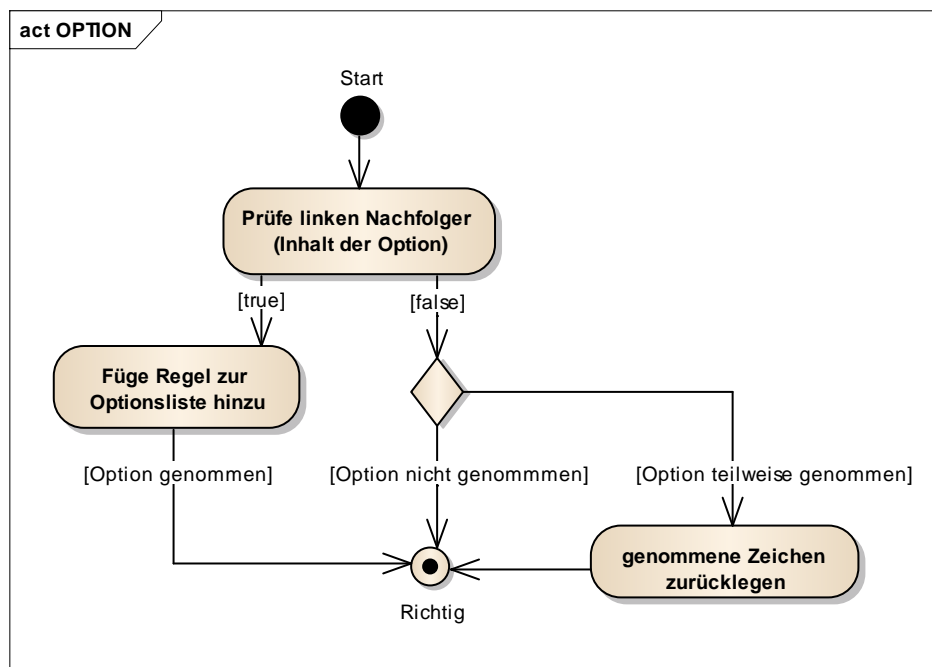


Abbildung 3.29: Aktivitätsdiagramm der Regel Option („OPTION“)

Die Optionalität [Abbildung 3.29] besitzt stets den Rückgabewert „Richtig“, denn sowohl eine einfach genommene Option, als auch eine übersprungene, sind semantisch korrekt. Falls die Optionsregel nicht gewählt wird muss der ursprüngliche Zustand des Eingabe-Stacks ggf. wiederhergestellt werden. Erfüllte Regeln werden zu einer Optionsliste hinzugefügt, da es in diesem Fall unter Umständen notwendig ist, genommene Zeichen nachträglich wieder auf den Stack „*textStapel*“ zurückzulegen.

Bei der Interpretation der in Abbildung 3.30 zu sehenden EBNF-Grammatik mit der Eingabe „7“ und Beginn bei der Produktion „*zwei*“ wird die einmalige Option „*[digit]*“ zunächst gewählt. Jedoch ist aufgrund des in Reihe folgenden „*digit*“ ein Zurücklegen unumgänglich. Mit der Startregel „*drei*“ würden sogar zwei Optionsregeln vorübergehend korrekt interpretiert werden, bis das eingehende Zeichen letztendlich wiederum dem letzten Nichtterminal der Reihenstruktur zugeordnet wird.

```
digit = "0" | "1" | ... | "9".  
zwei = [digit] digit.  
drei = [digit] [digit] digit.
```

Abbildung 3.30: *Grammatikbeispiel für Zurücklegen bei Optionen*

Mit Hilfe der Optionsliste werden die verschiedenen Kombinationsmöglichkeiten aller während des Interpreter-Ablaufs auftretenden Optionen geregelt. Diese wächst rasch mit der Menge der Optionalitäten an. Für das einfache Beispiel in Abbildung 3.31 gibt es bereits sieben unterschiedliche Wege der Zusammenstellung. Im tatsächlichen Programmablauf findet kein Zurücklegen der Symbole sondern ein Erproben aller Möglichkeiten statt. Sollte dabei keine passende Kombination gefunden werden, wird die entsprechende Nichtterminalregel (nicht die Option) als „Falsch“ gewertet.

```
test = [a] [b] [c].  
Kombinationsmöglichkeiten:  
a, b, c, ab, ac, bc, abc
```

Abbildung 3.31: *Beispiel für Kombinationsmöglichkeiten der Optionen*

f) Wiederholung

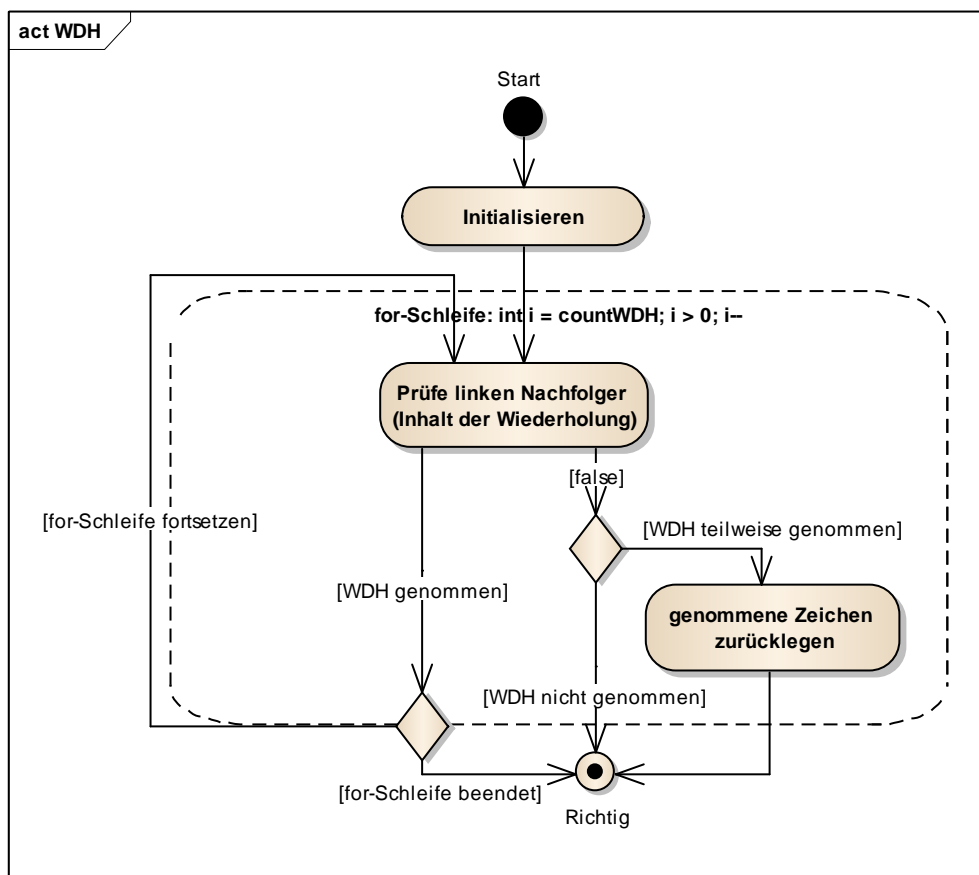


Abbildung 3.32: Aktivitätsdiagramm der Regel Wiederholung („WDH“)

Die Wiederholung besitzt, ebenso wie die Option, nur einen positiven Rückgabewert. Der Ablauf im Aktivitätsdiagramm [Abbildung 3.32] lässt sich ebenfalls vergleichen. Allerdings wird der Vorgang mit einer For-Schleife über die maximale Wiederholungsmöglichkeit („countWDH“), initial die Anzahl der Zeichen der Eingabe, wiederholt ausgeführt, da diese Art der Syntaxregel unendlich oft wiederholt auftreten kann.

```

digit = "0" | "1" | ... | "9".
number1 = {digit} digit.
number2 = digit {digit}.

```

Abbildung 3.33: Beispiel-Grammatik für „WDH“

Durch das Beispiel in Abbildung 3.33 wird deutlich, dass erneut auch das nachträgliche Zurücklegen eine spezielle Regelung erfordert. In der Produktion „number1“ würde die Wiederholung „{digit}“ bei der Eingabe „2707“ sämtliche Zeichen korrekt interpretieren, jedoch würde der Interpreter ohne Rückgabe der letzten „7“, aufgrund der in Reihe folgenden Regel „digit“, ein negatives Ergebnis zurückgeben. Bei der Definition „number2“ steht nach der Wiederholungsmöglichkeit nur das Endzeichen. In dieser Anordnung wird die „2“ von „digit“ und sämtliche übrige Zeichen von „{digit}“ erkannt.

Ein Zurücklegen ist also notwendig, wenn die Wiederholungsregel als linker Nachfolger in einer Reihenstruktur auftritt, weshalb sie in der Regel Reihe die in Abbildung 3.34 zu sehende gesonderte Behandlung erfährt. In der Anwendung findet wiederum keine wirkliche Rückgabe statt, sondern der Startwert „*countWDH*“ der abwärts zählenden For-Schleife aus der Regel Wiederholung [Abbildung 3.32] wird schrittweise dekrementiert. Mittels eines While-Konstrukts werden so alle Möglichkeiten nacheinander erprobt, um die passende Anzahl an Wiederholungen zu finden.

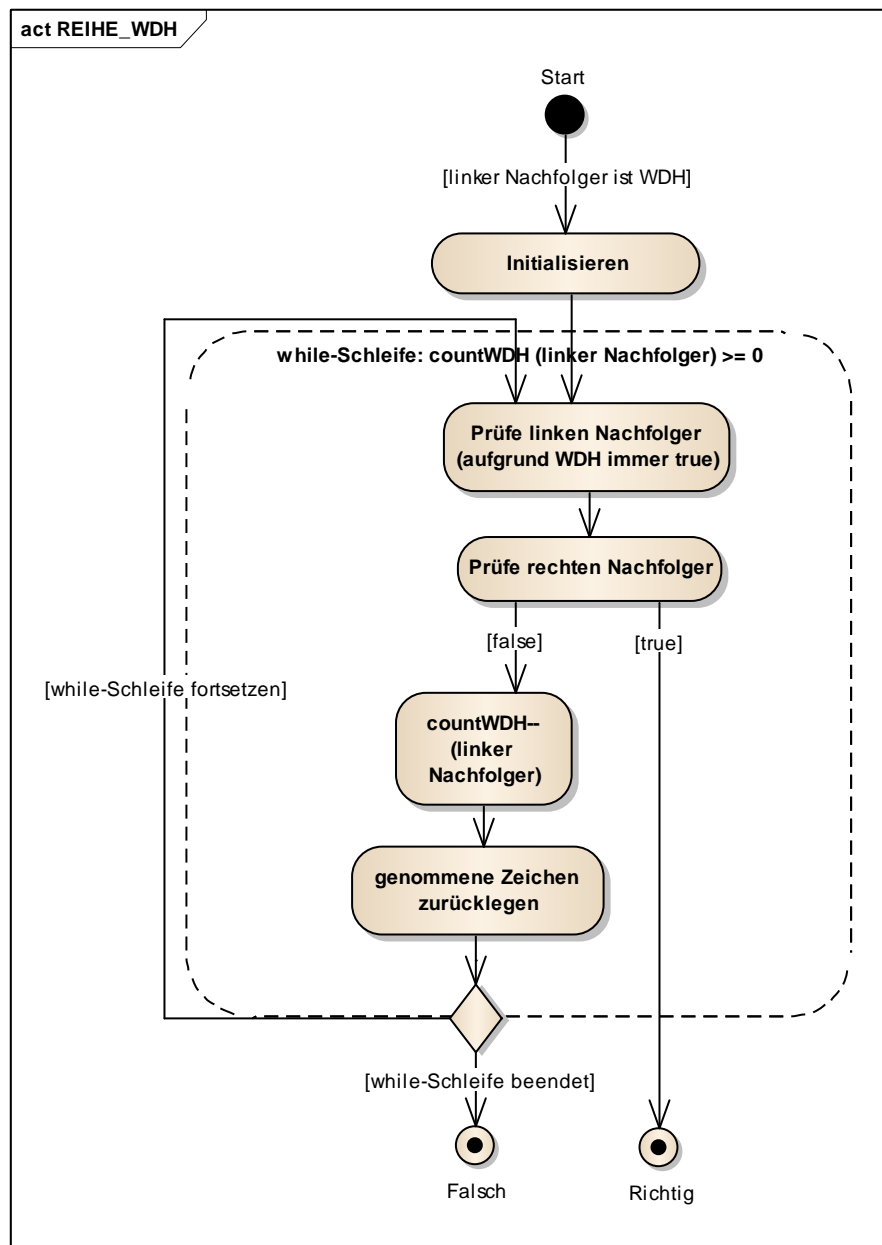


Abbildung 3.34: Aktivitätsdiagramm der Regel Reihe für den linken Nachfolgertyp „WDH“

g) Rekursion

Jegliche beim Interpretieren durchlaufene Regeln werden, wie in Abschnitt b) bereits erwähnt, zu einer Liste mit allen aufgerufenen Regeln hinzugefügt. Eine sog. Tiefenvariable speichert für jede Produktion die Anzahl der wiederholten bzw. rekursiven Durchläufe [Abbildung 3.26]. Dazu wird jeweils pro entsprechender Tiefe ein Objekt der Klasse „*ListElement*“ [Anhang 6.1.4] aus dem Package „*interpreter*“ instanziiert, in dem alle bis zu diesem Zustand genommenen Zeichen in einem Stack gelagert sind. So wird beim ggf. notwendigen Zurücklegen von Symbolen auf den Eingabestapel („*textStapel*“) verhindert, dass die aktuelle Tiefe Zugriff auf Zeichen aus höheren erhält.

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
number = digit number | digit.
```

Abbildung 3.35: Beispiel für rechtsseitige Rekursion

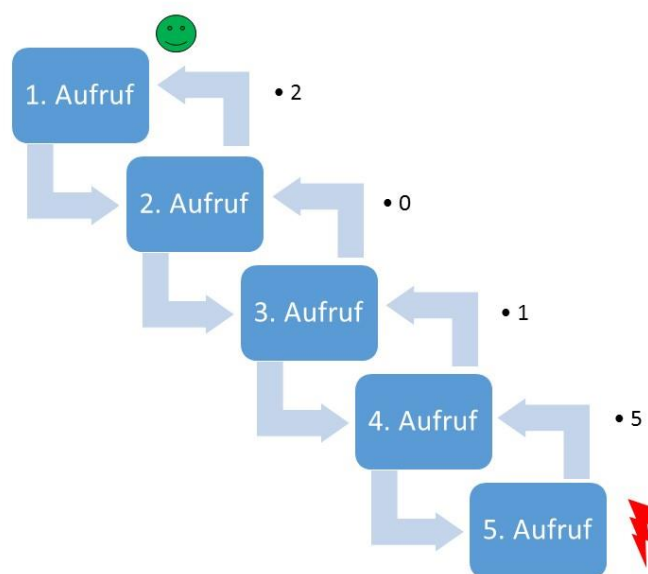


Abbildung 3.36: Interpretation der Eingabe „2015“ mit rekursiver Grammatikregel „*number*“ [Abbildung 3.35]

Rechtseitig rekursive Syntaxregeln können auf diese Weise korrekt interpretiert werden. Bei der Interpretation des Beispiels aus Abbildung 3.35 mit der Eingabe „2015“ und der Startregel „*number*“ wird die Produktion zunächst viermal aufgerufen und jeweils eine Ziffer der links stehenden Regel „*digit*“ zugeordnet. Beim fünften Aufruf ist der Eingabestapel leer und folglich wird der Wert „Falsch“ um eine Tiefe nach oben zurückgegeben. Dort stellt der Interpreter fest, dass die Reihe „*digit number*“ nicht zur Eingabe passt, legt die genomme „5“ kurzzeitig auf den Stack zurück und ordnet diese anschließend richtigerweise der

Alternative bzw. der Rekursionsabbruchbedingung „*digit*“ zu. Abschließend wird der Rückgabewert „Richtig“ nach oben zurückgereicht und der Interpretiervorgang erfolgreich beendet [Abbildung 3.36].

Die über mehrere Ebenen verschachtelte Rekursion wird durch die obige Realisierung ebenfalls richtig interpretiert. Als Beispiel hierfür dient die Selbstdefinition der EBNF [Abbildung 2.1], in der die Regel „*expression*“ nach Durchlauf der Produktionen „*term*“ und „*factor*“ sich selbst rekursiv aufruft.

Allerdings bedarf es einer separaten Lösung von mittig-rekursiven Syntaxdefinitionen. Diese werden durch den Parser in eine Struktur mit unendlichen Wiederholungen umgewandelt [Abbildung 3.37]. Abbildung 3.38 zeigt den Syntaxbaum vor und Abbildung 3.39 nach dieser Umwandlung. Um sicherzustellen, dass die Nichtterminale vor und nach der Rekursion die gleiche Anzahl an interpretierten Zeichen besitzen, wird geprüft, ob die beiden Wiederholungen genauso oft genommen werden. Diese Kontrolle ist bisher nur implementiert, falls die Startregel die mittige Rekursion enthält.

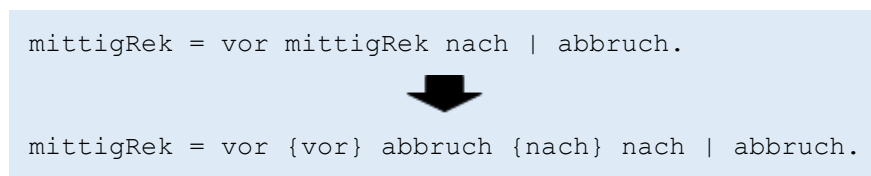


Abbildung 3.37: Umwandlung von mittig-rekursiven Regeln

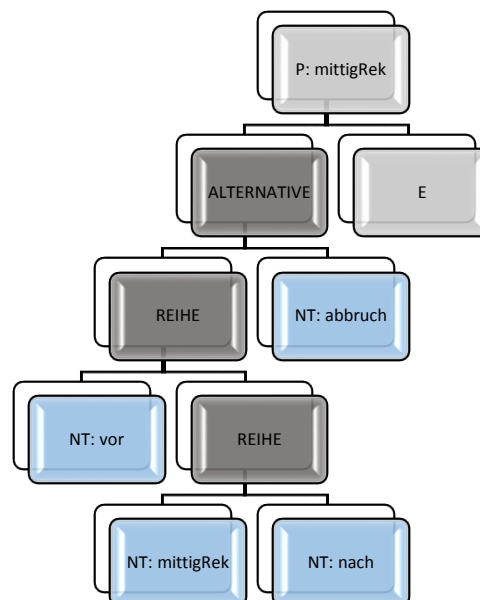


Abbildung 3.38: Syntaxbaum einer mittig-rekursiven Regel vor Umwandlung

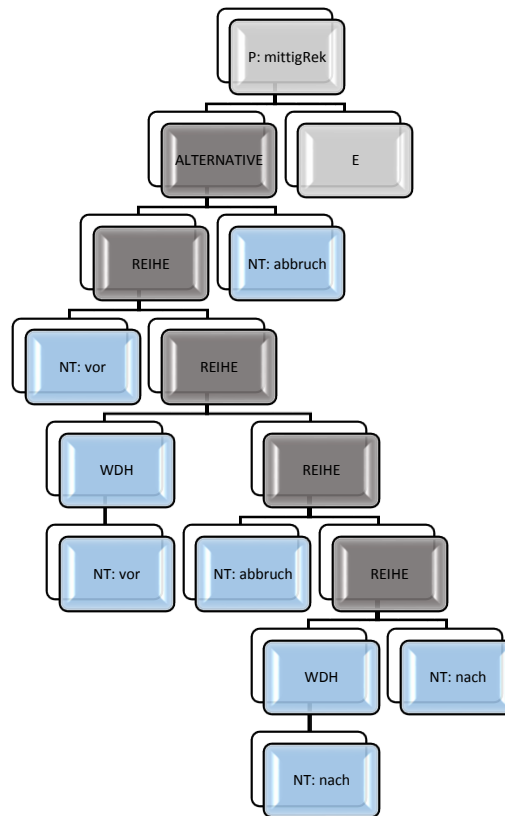


Abbildung 3.39: Syntaxbaum einer umgewandelten mittig-rekursiven Regel

Die Produktion „*numberMid*“ in Abbildung 3.40 interpretiert korrekterweise nur eine ungerade Anzahl an Ziffern als „Richtig“, wohingegen die Definition „*rule*“, die die mittig-rekursive Regel als Nichtterminal aufruft, diese Überprüfung nicht vornimmt und hier somit auch fälschlicherweise gerade Folgen von „*digit*“ in der Regel „*numberMid*“ möglich sind.

Linksseitige Rekursion wird in dieser Version der Anwendung „EBNF-Checker“ nicht unterstützt, weshalb derartige Regeldefinitionen bereits im Parser eine Meldung erzeugen [Kapitel 3.2.3c)]. Die Umsetzung könnte mit einer Umwandlung aller links- in rechtsrekursive Regeln erfolgen, nachdem die Detektion bereits realisiert ist.

```

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
numberMid = digit numberMid digit | digit.
rule = numberMid ";" .

```

Abbildung 3.40: Beispiel für mittig-rekursive Grammatikregeln

h) Fehlerbehandlung

Im CUP-Parser werden bereits mehrfache Definitionen einer Produktion behandelt. Außerdem werden Regeln mit linksseitiger Rekursion erkannt und deren Grammatik nicht zur Interpretation zugelassen, bevor keine Korrektur dieser, in „EBNF-Checker“ nicht unterstützter, Syntax erfolgt [Kapitel 3.2.3c)]. Weitere semantische Fehler werden in der Klasse „*Interpreter*“ bearbeitet und sind im folgenden Abschnitt ausgeführt.

Falls der Interpreter versucht, einen nicht in der HashMap „*treeMap*“ verfügbaren Syntaxbaum aufzurufen, wird dem Benutzer eine entsprechende Meldung angezeigt, wobei zwischen der fehlenden Definition der angegebenen Startregel und einer einfachen Nichtterminalregel unterschieden wird [Abbildung 3.41].

```
Error: Couldn't find rule "number"
Error: Couldn't find your start rule
```

Abbildung 3.41: Fehlermeldungen bei nicht deklarierten Regeln

Des Weiteren erscheint eine Warnung auf der Weboberfläche, falls die Tiefenvariable einer Produktion einen höheren Wert als 500 erreicht. Sollte die semantische Analyse eine Regel zu oft rekursiv aufrufen, wird das stetige Aufrufen der Funktion „*interpret*“ an dieser Stelle mit einem negativen Ergebnis unterbrochen [Abbildung 3.42]. So können durch Rekursion auftretende Endlosschleifen erkannt und abgebrochen werden. Zusätzlich wird damit eine obere Schranke für die korrekte rekursive Interpretation definiert.

```
if (tree.getRoot().getTiefe() > 500) {
    //Grenze für Rekursion
    interpreterAusgabe = "Warning: The rule \"" + regel.getValue() +
    "\" has been called to often recursively\n";
    return -1; //Falsch
}
```

Abbildung 3.42: Abbruch der Rekursion bei mehr als 500 Aufrufen in der Klasse „*Interpreter*“

Das Beispiel in Abbildung 3.43 zeigt den kritischen Fall einer über die Definitionen „*term*“ und „*exp*“ verschachtelten linksseitigen Rekursion, die durch den Parser nicht frühzeitig herausgefiltert wird und die Notwendigkeit dieser Warnung verdeutlicht.

Neben der soeben erläuterten Rekursionswarnung wird in der Grafik als letzte Fehlermeldung das negative Ergebnis der semantischen Analyse angezeigt. Dort wird dem Benutzer zusätzlich eine Fehlerkette mit der Reihenfolge des Rückgabewertes „Falsch“ bereitgestellt. Beginnend mit dem ersten Auftreten des negativen Interpretierergebnisses werden beim Zurückgeben des Wertes zur Startregel die Namen der durchlaufenen Produktionen ausgegeben. Als letztes Glied einer Kette ist also jeweils die angegebene Startregel zu finden. Um die Benutzerfreundlichkeit der Anzeige zu gewährleisten, ist diese Sequenz auf maximal 31 Produktionen (30 Syntax- und

eine Startregel) beschränkt, die in Abbildung 3.43 aufgrund des vielfach rekursiven Aufrufs vollständig ausgeschöpft werden.

```
EBNF-Grammatik (Startregel: "term"):  
term = exp.  
exp = term.  
  
Ausgabe:  
Result: CORRECT (parsed without errors)  
Warning: The rule "term" has been called to often recursively  
Error: Couldn't interpret your input (error sequence: --> exp --> term --  
> exp --> term --> exp --> term --> exp --> term --> exp --> term --> exp  
--> term --> exp --> term --> exp --> term --> exp --> term --> exp -->  
term --> exp --> term --> exp --> term --> exp --> term --> exp --> term  
--> exp --> term --> term)
```

Abbildung 3.43: Fehlermeldungen eines über zwei Ebenen linksrekursiven Grammatikbeispiels

```
EBNF-Grammatik (Startregel: "boolean", Eingabe: ""):  
boolean = false | true.  
false = "0".  
true = "1".  
  
Ausgabe:  
Result: CORRECT (parsed without errors)  
Error: Couldn't interpret your input (error sequence: --> false --> true  
--> true --> false --> boolean)
```

Abbildung 3.44: Fehlermeldung einer falschen Eingabe zu einer korrekten Beispielgrammatik

Abbildung 3.44 verdeutlicht dieses Prinzip, wobei in der Fehlerkette die Arbeitsweise des Interpreters deutlich zu erkennen ist. Zunächst wird die Alternative in der angegebenen („*false* ---> *true*“) und im anschließenden „zweiten Versuch“ in umgekehrter Reihenfolge („*true* ---> *false*“) geprüft.

Bei Interpretieren mit dem Ergebnis „Richtig“, ohne semantische Fehler, erhält der Anwender die in Abbildung 3.45 zu sehende Rückmeldung.

```
Result: CORRECT (intepreted without errors)
```

Abbildung 3.45: Korrektes Interpretieren ohne semantische Fehler

3.3 GWT-Weboberfläche

Die anschließenden Gliederungspunkte sollen einen guten Überblick über die Entwicklung der GWT-Weboberfläche zur Anwendung „EBNF-Checker“ vermitteln. Dabei werden nicht alle einzelnen Bestandteile detailliert beschrieben. Es wird vorrangig auf diverse Problemstellungen, die während der Programmierung auftreten, wertgelegt. Das entstandene Ergebnis wird in Kapitel 4.2 erläutert und getestet.

3.3.1 Entwicklungszyklus

Aufgrund der Architektur des Dokumentenservers der Universität der Bundeswehr München, der nicht die Funktion eines Webservers übernehmen kann, und des Ziels dieser Arbeit, eine Lernsoftware für Studenten zu entwickeln, wird die GWT-Anwendung „EBNF-Checker“ nur clientseitig realisiert. Dieses für das Google Web Toolkit eher untypische Verfahren hat jedoch den Vorteil, dass der Benutzer keine Daten mit dem Server austauscht, sondern sich das Programm dort lediglich herunterlädt. Der Student bzw. die Studentin muss somit beim Üben nicht befürchten, dass der Dozent bzw. die Dozentin ihre Eingaben abspeichert und auswertet. Zudem lässt sich „EBNF-Checker“ so offline verwenden. Einige Nachteile, die diese Entwicklung aber mit sich zieht, werden in den jeweiligen Abschnitten angesprochen.

a) Allgemeine Projektstruktur

Ein GWT-Projekt besteht immer aus einem Quellverzeichnis, in das der Java-Quellcode geschrieben wird, und einem Webverzeichnis mit dem durch den GWT-Compiler generierten Code der GWT-Applikation. Bei einer ausschließlich clientseitigen Realisierung stellt „*EBNF_CheckerEntryPointClass*“ eine Art Main-Klasse und die überschriebene Funktion „*onModuleLoad*“ eine Main-Funktion dar. Bei der Verwendung von weiteren Ressourcen dienen sog. „GWT-Modules“ („*EBNF_Checker.gwt.xml*“) zur Verknüpfung, die in einem von Google etwas abgewandelten XML geschrieben werden [33].

b) Widgets

Grafikobjekte, wie z.B. Tasten, Schaltflächen oder Texteingabefelder, nennt man in GWT „Widget“. Einen kleinen Überblick bietet [34].

„EBNF-Checker“ benötigt einerseits „*Buttons*“ zum Start der Ausführung von Aktionen (z.B. Parsen und Interpretieren), deren Verhalten beim Anklicken mit sog. „*ClickHandlers*“ geregelt wird. Um beim Parsen und Interpretieren Doppelklicks auf die Tasten zu vermeiden und den Analysevorgang damit zu stören, werden diese Bedienelemente zu Beginn der Aktion ausgegraut (deaktiviert) und erst nach abgeschlossener Untersuchung der Eingaben wieder aktiviert.

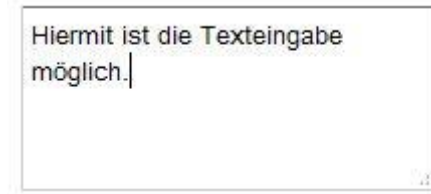


Abbildung 3.46: „TextArea-Widget“

Für die Erfassung der EBNF-Grammatik, der dazu getätigten Eingaben und der Startregel wird der GWT-Standardtexteditor „*TextArea*“ verwendet [Abbildung 3.46].

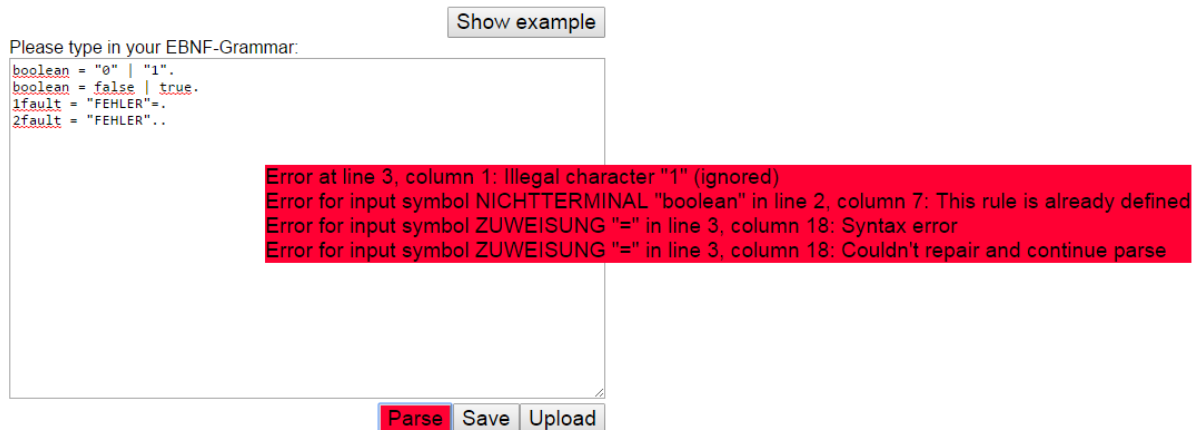


Abbildung 3.47: Parsen einer fehlerhaften Grammatik mit entsprechenden Meldungen

Ergebnisse der lexikalischen, syntaktischen und semantischen Analyse werden mit sog. „*DecoratedPopupPanels*“ dargestellt, die jeweils in Bildschirmmitte erscheinen, den Benutzer über den Erfolg (grün) oder das Fehlschlagen (rot) eines Vorgangs informieren und mit einem Klick außerhalb des Fensters wieder verschwinden. Abbildung 3.47 zeigt die Darstellung auf der Weboberfläche beim Parsen des fehlerhaften Grammatikbeispiels aus Kapitel 3.2.3c) [Abbildung 3.19].

Ein weiterer Aspekt, den es zu betrachten gilt, ist die Formatierung von Textzeichen. Da die Textdarstellung auf der Weboberfläche in HTML erfolgt, müssen Java-Strings entweder konvertiert oder bereits mit HTML-Syntax gefüllt sein, um beispielsweise Zeilenumbrüche und aufeinander folgende Leerzeichen korrekt darzustellen, die u.a. bei der Syntaxbaumausgabe von Nöten sind [Abbildung 3.48].

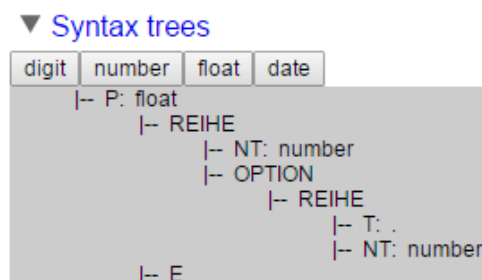


Abbildung 3.48: Beispiel für Syntaxbaumausgabe

c) Panels

```
final RootPanel rootPanel = RootPanel.get();
```

Abbildung 3.49: Erzeugen des Attributs „rootPanel“

Die einzelnen „Widgets“ werden in „EBNF-Checker“ mit sog. „Panels“ auf der Oberfläche angeordnet, die beim Google Web Toolkit u.a. für die Platzierung von Objekten hilfreich sind. Eine kleine Auswahl lässt sich erneut in [34] finden. Beispielsweise lassen sich Abschnitte mit einem „*HorizontalPanel*“ horizontal und mit einem „*VerticalPanel*“ vertikal unterteilen. Die gesamte Webseite wird mittels „*RootPanel*“ angesprochen [Abbildung 3.49]. Jegliche sichtbaren Elemente werden dort direkt oder mit entsprechenden „Panels“ hinzugefügt.

d) Ahome-Client-IO

Als Zusatzfunktion kann es für Benutzer nützlich sein, EBNF-Grammatiken in Textdateien auf dem Computer zu speichern und aus diesen auch Grammatikeingaben hochzuladen. Zur Umsetzung der Anwendungsfälle „EBNF-Grammatik speichern“ und „EBNF-Grammatik hochladen“ aus Abbildung 3.2 werden in Java einige Klassen aus dem „*java.io*“-Package (z.B. *File*) benötigt. Dabei bedarf es also mehr als der vom Google Web Toolkit unterstützten Emulationen [27]. Deshalb ist das lokale Speichern und Hochladen von Textdateien in GWT clientseitig nicht ohne spezielle Hilfsmittel möglich.

Die „*Ahome-Client-IO*“ [35] stellt u.a. diese Funktionalität zur Verfügung. Allerdings wird für deren Verwendung mindestens die GWT-Version 2.5.0 und „Adobe Flash Player 11“ vorausgesetzt.

Zunächst wird die GWT-Bibliothek „*ahome-client-io-2.0.0.jar*“ in das Projekt eingebunden und zu Beginn der Methode „*onModuleLoad*“ initialisiert [Abbildung 3.50]. Nach Anlegen entsprechender Bedientasten kann die eingegebene EBNF-Grammatik mit dem in Abbildung 3.51 zu sehenden Befehl in eine Textdatei gespeichert werden. Abbildung 3.53 zeigt die Darstellung dieses Vorgangs auf der Weboberfläche. Nach dem Klicken auf das Element „*Save*“ wird ein graues Popup am oberen Ende der Webseite sichtbar, dessen Betätigung noch erforderlich ist, um das „*Speichern-unter*“-Fenster zu öffnen. Das Prinzip des Hochladens erfolgt identisch. Die zugehörigen Befehle sind in Abbildung 3.52 zu sehen.

```
//Initialisierung der ClientIO  
ClientIO.init();
```

Abbildung 3.50: Initialisierung *ClientIO*

```
//buttonSpeichern-Aktion ("Save")
buttonSpeichern.addClickListener(new ClickHandler() {
    @Override
    public void onClick(ClickEvent event) {
        //speichert den aktuellen Inhalt der EBNF-Grammatikeingabe
        ClientIO.saveFile(textGrammatik.getText(), "file.txt");
    }
});
```

Abbildung 3.51: *Speichern der EBNF-Grammatik in Textdatei*

```
//buttonLaden-Aktion ("Upload")
buttonLaden.addClickListener(new ClickHandler() {
    @Override
    public void onClick(ClickEvent event) {
        ClientIO.addFileSelectHandler(new ClientIoFileSelectHandler() {
            @Override
            public void onFileLoaded(String fileName, String fileType,
                ByteArray data, double fileSize) {
                //Text aus Datei in die Grammatikeingabe setzen
                textGrammatik.setText(data.readMultiByte(fileSize, "UTF-8"));
            }

            @Override
            public void onCancel() {
            }

            @Override
            public void onIoError(String errorMessage) {
            }
        });
        //Popup zum Auswählen der Datei öffnen
        ClientIO.browse();
    }
});
```

Abbildung 3.52: *Hochladen einer EBNF-Grammatik aus einer Textdatei*

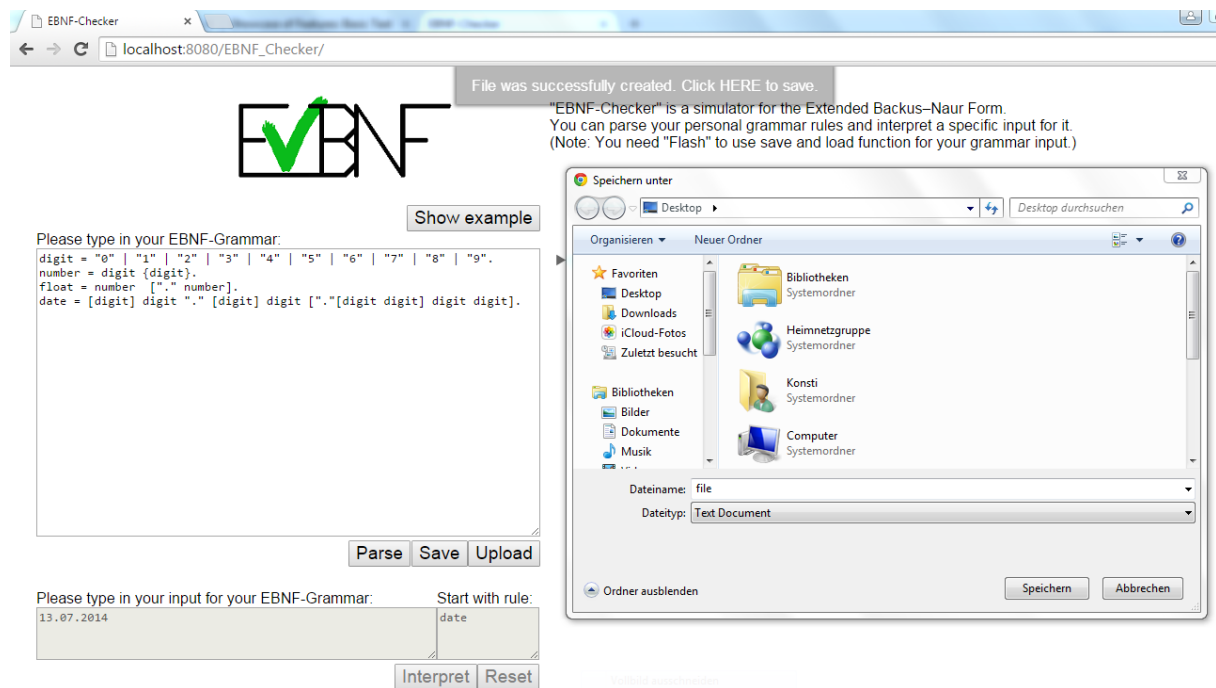


Abbildung 3.53: Darstellung des Speichervorgangs mit ClientIO auf der Weboberfläche

e) Design

```
//Anlegen eines Buttons "button" und Zuweisen der ID "button-id"
final Button button = new Button("Click me!");
DOM.setAttribute(button.getElement(), "id", "button-id");
```

Abbildung 3.54: Anlegen eines Button-Widgets und Zuweisen einer eindeutigen ID

Für das Design, die Anordnung und die farbliche Gestaltung der Elemente auf der Weboberfläche bieten sich CSS-Dateien an. In der Datei „*myStylesheet.css*“ wird für einige GWT-Klassen der Stil angepasst, aber es können auch einzelne Objekte bestimmter „Widgets“ individuell bearbeitet werden, nachdem sie durch die Zuweisung einer eindeutigen ID [Abbildung 3.54] ansprechbar sind [Abbildung 3.55]. Unveränderte Attribute behalten die vom Google Web Toolkit festgelegten Standardwerte.

```
/*gültig für jedes Objekt der entsprechenden Klasse*/
.gwt-Button {
    font-family: Roboto,sans-serif; /*Schriftart*/
    font-size: 120%;                /*Schriftgröße*/
}

/*gültig für ein Objekt mit entsprechender ID*/
#button-id {
    margin-bottom: 25px;             /*Abstand nach unten*/
    background-color: #FF0033;      /*Hintergrundfarbe*/
}
```

Abbildung 3.55: Anpassen des Designs von GWT-Klassen und GWT-Objekten mit CSS

Das Einbinden der CSS-Datei erfolgt zu Beginn der Methode „*onModuleLoad*“ der Klasse „*EBNF_CheckerEntryPoint*“ über das Interface „*Resources*“ [Anhang 6.1.5]. So wird im Fehlerfall ein Compilerfehler ausgegeben [Abbildung 3.56].

```
Resources.INSTANCE.css().ensureInjected();
```

Abbildung 3.56: Einbinden der CSS-Datei „*myStylesheet.css*“

3.3.2 Integration der Java-Applikation

Nachdem in den vorausgehenden Abschnitten vorwiegend der Entwurf der grafischen Komponenten dargelegt wurde, wird nun die Integration des „EBNF-Simulators“ [Kapitel 3.2] in die GWT-Anwendung und der Zugriff auf diesen beschrieben.

Dieser erfolgt über die Klasse „*Interpreter*“ [Abbildung 3.23]. Die Grammatik und die Eingabe werden als String mit der Funktion „*setGrammatik*“ und „*setEingabe*“ übergeben. Der Parsevorgang wird mit „*initParsing*“ gestartet und dessen Meldungen mit „*getLexParseAusgabe*“ als String ausgelesen. Beim Interpretieren geschieht dies analog mit „*initInterpreting*“ unter Übergabe einer Startregel als Parameter und „*getInterpreterAusgabe*“. Mit den Booleschen-Methoden „*isSyntaxError*“ und „*isInterpreterError*“ kann die GWT-Anwendung zwischen dem Erfolg (grüne Anzeige) und dem Fehlschlagen (rote Anzeige) einer Analyse unterscheiden.

Die Java-Klassen „*Scanner*“, „*Parser*“ und „*Interpreter*“ benötigen den Import folgender nicht mit GWT kompatibler Dateien aus der CUP-Laufzeitbibliothek „*java-cup-11b-runtime.jar*“ und der JRE:

```
import java_cup.runtime.Symbol;  
import java_cup.runtime.ComplexSymbolFactory;  
import java_cup.runtime.ComplexSymbolFactory.Location;  
import java.io.StringReader;
```

Abbildung 3.57: Nicht unterstützte Importe des „EBNF-Simulators“

Es bedarf also einer Lösung, um den „EBNF-Simulator“ dennoch im Clientbereich der Applikation verwenden zu können. Deshalb wird ein Package „*lib*“ mit einem „GWT-Module“ „*Library.gwt.xml*“ angelegt, das mit dem Hauptmodul „*EBNF_Checker.gwt.xml*“ verknüpft ist. Mittels der Instruktion in Abbildung 3.58 wird die Package-Deklaration aller unter „*lib*“ liegenden Dateien vom GWT-Compiler überschrieben. Beispielsweise kann so eine individuelle Beschreibung für die in der Java-Laufzeitumgebung bereits existierende Klasse „*StringReader*“ aus „*java.io*“ programmiert werden. Als Package wird „*java.io*“ angegeben, obwohl die Datei in einem Unterordner von „*lib*“ liegt. Die von der Entwicklungsumgebung angezeigte Meldung, diese Angabe sei nicht korrekt, wird ignoriert. So tritt beim Kompilieren nicht der Fehler auf, dass das Google Web Toolkit diese Datei nicht unterstützt, sondern GWT

ersetzt die Klasse aus der Java-Bibliothek mit der neuen Definition. Dieses Vorgehen wird als Emulation bzw. Nachahmung bezeichnet. Die Herausforderung besteht nun darin, eine entsprechende Nachbildung zu entwickeln. Falls die einzubindenden Java-Dateien erneut GWT-kritische Importe besitzen, müssen auch für diese Emulationen gefunden werden [36].



```
<super-source path="" />
```

Abbildung 3.58: „Super-Source“

Für die benötigten Klassen „*Reader*“ und „*StringReader*“ (erbt von „*Reader*“) aus „*java.io*“ werden in diesem Programm bereits vorhandene GWT-Nachahmungen [37] verwendet, deren Funktionen für die Verwendung in „EBNF-Checker“ genügen.

Für die CUP-Importe werden sieben Java-Dateien⁴ des CUP-Quellcodes [25] in ein entsprechend neu angelegtes Package „*lib.java_cup.runtime*“ kopiert. Allerdings müssen dort für die GWT-Kompatibilität noch einige Anpassungen vorgenommen werden. Diese Änderungen zeigen ausschließlich bei der Fehlerausgabe des Parsers negative Auswirkungen. Neben der in Kapitel 3.2.3c) beschriebenen Meldung unterstützt der CUP-Parser bei einem Syntaxfehler zusätzlich noch die Angabe einer Liste von Token, die anstatt des fehlerhaften Symbols als Eingabe korrekt wären. Dazu ist der Import von „*java.lang.reflect.Field*“ in der Klasse „*lr_parser*“ unumgänglich, den das Google Web Toolkit mit der in dieser Arbeit verwendeten Version nicht zulässt. Zudem war es nicht möglich, eine Emulation zu erstellen bzw. zu verwenden, weshalb diese Art Fehlermeldung in „EBNF-Checker“ nicht unterstützt wird.

Nachdem Scanner, Parser und Interpreter in einem „EBNF-Simulator“ zusammengefasst werden, ist mit der GWT-Anwendung „EBNF-Checker“ zusätzlich eine webbasierte Benutzeroberfläche entstanden, die die Funktionalität des Simulators beinhaltet.

⁴ *ComplexSymbolFactory*, *DefaultSymbolFactory*, *lr_parser*, *Scanner*, *Symbol*, *SymbolFactory*, *virtual_parse_stack*

4 Evaluation

4.1 Test des EBNF-Simulators

Die nachfolgenden drei Test-Grammatiken mit entsprechenden Eingaben dienen dem Aufzeigen der Möglichkeiten und Grenzen der entwickelten Anwendung.

4.1.1 Selbstdefinition der EBNF

Im ersten Testfall erhält der „EBNF-Simulator“ als Grammatik die Selbstdefinition der EBNF [7], wobei ein „*character*“ zur Vereinfachung nur aus einem Buchstaben oder einer Ziffer bestehen darf. Diese, in Abbildung 4.1 dargestellte Syntax, wird durch das Programm korrekt geparkt und einfache (Eingabe 1), sowie komplexere Eingaben (Eingabe 2), werden mit der Startregel „*syntax*“ richtig interpretiert [Abbildung 4.2].

```
syntax = { production }.
production = identifier "=" expression ".".
expression = term { "|" term }.
term = factor { factor }.
factor = identifier | string | "(" expression ")" | "[" expression "]" |
        "{" expression "}".
identifier = letter { digit | letter }.
string = ""{character}"".
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k"
        | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" |
        "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
        "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
        "V" | "W" | "X" | "Y" | "Z".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
character = digit | letter.
```

Abbildung 4.1: Beispiel 1 für Grammatikeingabe: EBNF-Selbstdefinition [7]

Dabei wird die Regel „*expression*“ vielfach über mehrere Ebenen rekursiv durchlaufen. Außerdem werden beim Interpretieren Wiederholungs-, Reihen- und Alternativregeln angewendet. Zu beachten ist jedoch, dass die Eingabe exakt so auszusehen hat, wie in der Grammatik beschrieben. Zusätzliche „Whitespaces“, wie beispielsweise in Eingabe 3

[Abbildung 4.2], werden vom Interpreter, anders als beim Parser, ausgewertet und führen zu einem negativen Ergebnis.

```
Eingabe 1:
boolean=true|false.

Eingabe 2:
regel=(["a"|"b"]test1{test2|test3})|"TERMINAL".regel2={test4
[true|false]}.number=digit{digit}["PUNKT"digit{digit}].

Eingabe 3:
number = digit { digit } .
```

Abbildung 4.2: Eingaben zu Beispiel 1 für Grammatikeingabe [Abbildung 4.1]

4.1.2 Integer-Literale in Java

Um die, im obigen Beispiel nicht verwendeten, Elemente der Optionalität und Klammerung der EBNF zu testen, werden folgende EBNF-Regeln für Integer-Literale in Java [38] definiert:

```
intLiteral = ["+" | "-"](octLiteral | decLiteral | hexLiteral)["l" | "L"].
octLiteral = "0" octDigit { octDigit }.
octDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7".
decLiteral = digit1 { digit } | "0".
hexLiteral = ("0x" | "0X") hexDigit { hexDigit }.
hexDigit = digit | "a" | "A" | "b" | "B" | "c" | "C" | "d" | "D" | "e" |
"E" | "f" | "F".
digit = "0" | digit1.
digit1 = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Abbildung 4.3: Beispiel 2 für Grammatikeingabe: Integer-Literale in Java [38]

Nach erfolgreichem Parsen kann der Interpreter erneut verschiedene Eingaben zur Startregel „*intLiteral*“ semantisch korrekt deuten, wobei er dort und in der Produktion „*hexLiteral*“ auch Optionen und Klammern durchläuft.

```
Eingabe 1: -2015L
Eingabe 2: 0xFFAB
Eingabe 3: -0171
```

Abbildung 4.4: Eingaben zu
Beispiel 2 für Grammatikeingabe
[Abbildung 4.3]

4.1.3 MiniB

Die Syntax der, aus der Vorlesung „Programmerzeugungssysteme“ bekannten, Sprache MiniB [39] ist in Abbildung 4.5 beschrieben. Diese sehr komplexe EBNF-Grammatik lässt sich wiederum ohne Syntaxfehler parsen, jedoch stößt die Anwendung „EBNF-Simulator“ beim Interpretieren eines MiniB-Beispiels zur Berechnung der Fakultät [40] an ihre Grenzen.

```

program = { definition }.
definition = identifier "(" [parameter { "," parameter } ] ")" "{"
expression ";" "}".
parameter = identifier.
expression = condition.
condition = sum | comparisson [ "?" expression ":" expression ].
comparisson = sum CompareOp sum.
CompareOp = "<" | "<=" | "==" | "!=" | ">=" | ">".
sum = term { ("+" | "-") term }.
term = factor { ("*" | "/" | "%") factor }.
factor = ["+" | "-"] number | parameter | functionCall | "(" expression
").".
functionCall = identifier "(" [expression { "," expression } ] ")".
number = digit {digit} [ "." digit {digit} ].
identifier = letter { letter }.
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k"
| "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" |
"x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
"J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" |
"V" | "W" | "X" | "Y" | "Z".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

```

Abbildung 4.5: Beispiel 3 für Grammatikeingabe: MiniB [39]

Mit der Startregel „*program*“ gibt der Interpreter für die Eingabe der Fakultäts- und anschließender Mainfunktion aus Abbildung 4.6 (Eingabe 1) fälschlicherweise eine Fehlermeldung aus. Allerdings sind Eingabe 2 und Eingabe 3 zu den Regeln „*definition*“ und „*program*“, ebenso wie Eingabe 4 zu „*program*“, richtig interpretierbar.

Der Grund für dieses Fehlverhalten ist die äußerst komplexe Verschachtelung der Options- und Wiederholungsregeln („*definition*“, „*condition*“ und „*functionCall*“), die durch die Regel „*expression*“ zusätzlich mehrfach rekursiv aufgerufen werden. Dabei steigt die Komplexität des Zurücklegens von temporär erkannten Symbolen auf den Eingabestapel exponentiell.

```

Eingabe 1:
fakultaet(n){n<=1?1:n*fakultaet(n-1);}main(i){fakultaet(i);}

Eingabe 2:
fakultaet(n){n<=1?1:n*fakultaet(n-1);}

Eingabe 3:
main(i){fakultaet(i);}

Eingabe 4:
fakultaet(n){fak(n-1);}main(i){fakultaet(i);}

```

Abbildung 4.6: Eingaben zu Beispiel 3 für Grammatikeingabe [Abbildung 4.5]

4.2 Weboberfläche

EBNF-Checker

"EBNF-Checker" is a simulator for the Extended Backus–Naur Form.
You can parse your personal grammar rules and interpret a specific input for it.

Show example (1)

Please type in your EBNF-Grammar:

(2)

Parse (3) Save (4) Upload (5)

Please type in your input for your EBNF-Grammar: (6)

Start with rule: (7)

Interpret (8) Reset (9)

Reset all (10)

Universität der Bundeswehr München

Developed by Konstantin Klinger (2015)
For feedback, bug and issue support: konstantin.klinger@unibw.de

About (11)

Abbildung 4.7: Startoberfläche „EBNF-Checker“

Die Bedienung der Webanwendung „EBNF-Checker“ ist einfach und benutzerfreundlich gestaltet. Der Zugriff durch den Benutzer erfolgt über einen einfachen HTML-Link, der „EBNF-Checker“ in beliebigen Browsern öffnet. In Abbildung 4.7 ist der Startbildschirm der Weboberfläche abgebildet.

Zunächst kann ausschließlich eine EBNF-Grammatik eingegeben (2) werden. Erst nach erfolgreichem Parsen (3) ist der Start des Interpretiervorgangs (8) möglich und die Textfelder für die Angabe der Eingabe (6) und der Startregel (7) zur geparsen Syntax sind beschreibbar. Diese können durch Klicken auf „Reset“ (9) einzeln oder zusammen mit der eingegebenen Grammatik durch „Reset all“ (10) zurückgesetzt werden. Dazu wird für Webbrowser unter den in Kapitel 3.3.1d) genannten Voraussetzungen das Speichern (4) und Laden (5) der EBNF-Grammatik in bzw. aus einer Textdatei unterstützt. Außerdem können weitere Informationen über das Programm angezeigt werden (11), die in Abbildung 4.8 zu sehen sind.

Die für unerfahrene Benutzer ausgelegten Beispieleingaben (1) dienen in Abbildung 4.9 zur Visualisierung einer korrekt abgelaufenen lexikalischen, syntaktischen und semantischen Analyse. Dort ist außerdem die Auswahlmöglichkeit zur Anzeige der erstellten Syntaxbäume zu sehen, die nach fehlerlosem Parsen erscheint. Der Anwender sieht dabei die optimierten Syntaxbäume so, wie der Interpreter sie auswertet. Mittige Rekursionen sind somit auch in umgewandelter Form zu sehen.

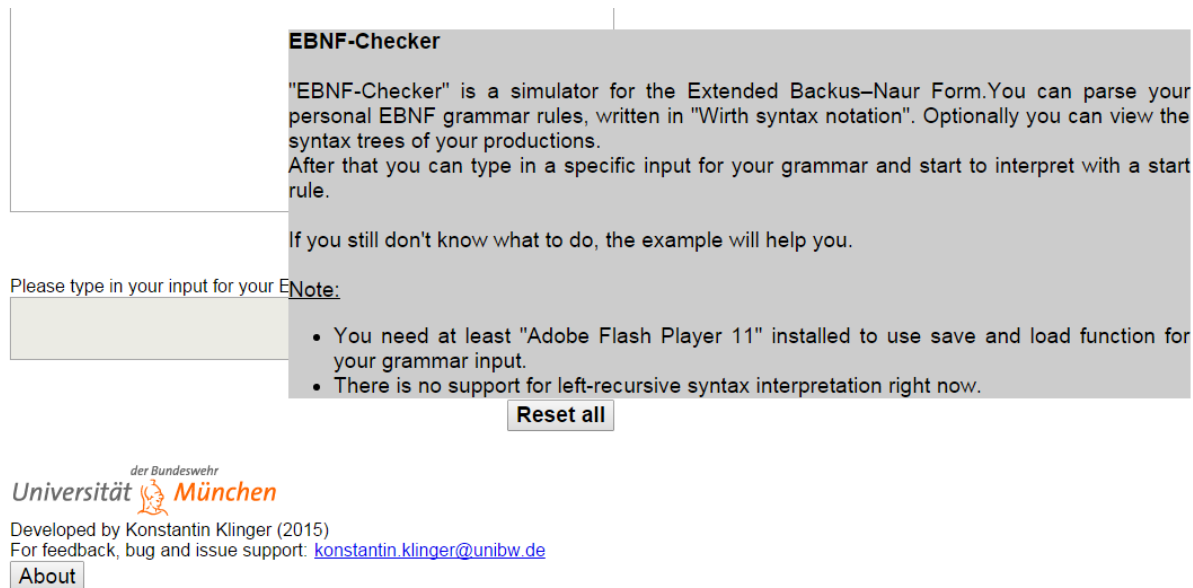



Abbildung 4.8: Popup-Oberfläche nach Klicken auf „About“ (Hilfe anzeigen lassen)



"EBNF-Checker" is a simulator for the Extended Backus–Naur Form.
You can parse your personal grammar rules and interpret a specific input for it.

[Show example](#)

Please type in your EBNF-Grammar:

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
number = digit {digit}.
float = number ["." number].
date = [digit] digit "." [digit] digit ["." [digit digit] digit digit].
```

Parse
Save
Upload

▼ Syntax trees

digit	number	float	date
-- P: date			
-- REIHE			
-- OPTION			
-- NT: digit			
-- REIHE			
-- NT: digit			
-- REIHE			
-- T: .			
-- REIHE			
-- OPTION			
-- NT: digit			
-- REIHE			
-- NT: digit			
-- OPTION			
-- REIHE			
-- T: .			
-- REIHE			
-- OPTION			
-- REIHE			
-- NT: digit			
-- NT: digit			
-- REIHE			
-- NT: digit			
-- NT: digit			
-- E			

Please type in your input for your EBNF-Grammar:

13.07.2014

Start with rule:

date

Interpret
Reset

Abbildung 4.9: Weboberfläche nach Parsen und Interpretieren des Beispiels aus „Show example“

Beim Testen der in Abschnitt 4.1 beschriebenen Grammatiken und Eingaben in der GWT-Webanwendung benötigt der Interpreter vereinzelt länger, um das Ergebnis der semantischen Analyse anzuzeigen. In manchen Fällen kann der Webbrowser deshalb die Meldung ausgeben, dass die Anwendung nicht mehr reagiert, da er während des Interpretierens keine Antwort des Programms erhält. Nach entsprechender Wartezeit erscheint auf der Oberfläche schließlich die Ausgabe des Interpreters. Dies hat den Grund, dass GWT-Applikationen clientseitig auf 4KB pro Cookie und 20 Cookies pro Domain speicherbegrenzt sind [41].

5 Zusammenfassung

5.1 Fazit

Die Arbeit zeigt wie auf Grundlage der Selbstdefinition der EBNF nach Wirth [7] mit den Generatoren JFlex [18] und CUP [25] ein funktionsfähiger EBNF-Parser geschaffen und dazu zunächst ein Parse-Baum erzeugt wird. Der anschließend optimierte Syntaxbaum wird von einem Interpreter während der semantischen Analyse überprüft. Dieser „EBNF-Simulator“ wird in eine GWT-Webanwendung mit Benutzeroberfläche integriert.

Das so entstandene Programm „EBNF-Checker“ erlaubt es in beliebigen Webbrowsern u. a. selbstdefinierte EBNF-Grammatiken zu parsen, deren Syntaxbäume anzeigen zu lassen und dazu getätigte Eingaben zu interpretieren. Der Zugriff erfolgt über einen einfachen HTML-Link.

Allerdings ist die Funktionalität des Interpreters mit zunehmender Komplexität der Syntax für einzelne Eingaben eingeschränkt. Durch mehrfache Verschachtelung von Wiederholungs-, Options- und Rekursionsregeln ineinander wächst die Anzahl der Zuordnungsmöglichkeiten einer Eingabe zu einer Grammatik exponentiell und ein nachträgliches Zurücklegen von bereits interpretierten Zeichen ist vermehrt erforderlich.

Jedoch erfüllt „EBNF-Checker“ dennoch für Beispiele aus der Vorlesung „Programm-erzeugungssysteme“ den angestrebten Zweck und bietet dem unerfahrenen Studenten eine geeignete Plattform, um die Kenntnisse der Extended-Backus-Naur-Form zu intensivieren.

5.2 Ausblick

Mit „EBNF-Checker“ ist unter den gegebenen Voraussetzungen eine sehr gelungene Applikation entstanden, deren Nutzen durch eine Erweiterung und Optimierung in einigen Aspekten noch gesteigert werden könnte.

Einerseits sollte im Interpreter zunächst die fehlende Implementierung von linksseitigen und mittigen Rekursionsregeln ergänzt bzw. vervollständigt werden. Andererseits bedarf es einer Optimierung des Zurücklegens von bereits interpretierten Zeichen, um auch sämtliche Eingaben zu komplexen Grammatiken korrekt verarbeiten zu können.

Zusätzlich könnte der Interpreter um die optionale Funktion des Ignorierens von „Whitespaces“ erweitert werden, um die Eingabe zu einer EBNF-Grammatik komfortabler zu gestalten. Dies

könnte mit einer Auswahlmöglichkeit auf der Weboberfläche oder mit einer Konvention über den groß- oder kleingeschriebenen Beginn von Produktionen erfolgen.

Außerdem könnte die aktuellste Version des Google Web Toolkit in die Anwendung integriert werden, was aufgrund der verwendeten Entwicklungsumgebung bisher nicht möglich ist [Kapitel 6.2]. So könnte der CUP-Parser bei einem Syntaxfehler eine Liste mit Symbolen ausgeben, die anstatt des fehlerauslösenden Zeichens zu erwarten wären. Hierfür wird von GWT ab Version 2.7.0 das Interface „*JField*“ [42] als Ersatz für „*java.lang.reflect.Field*“ angeboten.

Dazu könnte ein Editor für die Texteingaben auf der Weboberfläche verwendet werden, der die Angabe von Zeilennummern unterstützt und automatische Zeilenumbrüche als neue Zeilen erkennt. Überdies wären farbige Markierungen der lexikalischen und syntaktischen Fehler in der eingegebenen Grammatik wünschenswert.

Störend ist zudem die Notwendigkeit eines installierten und aktivierten „Flash“-Players, um die zu parsende Syntax speichern bzw. aus einer Textdatei hochladen zu können. Hier besteht die Möglichkeit den clientseitigen Zugriff auf Dateien in GWT mit HTML5 zu lösen, um diese Funktion dauerhaft nutzen zu können und eine mögliche Integration in die „Virtual-C IDE“ [43] vorzubereiten.

Ferner könnte der sog. „HTML5 Storage“ [41] des GWT die teilweise längere Wartezeit auf das Ergebnis des Interpreters beachtlich verkürzen, da diese Technologie die Speicherkapazität des Clients deutlich erhöhen kann (5MB pro Domain, unbegrenzter Sitzungsspeicher) [41].

Des Weiteren wäre ein Testlauf des entwickelten Programms sinnvoll, der beispielsweise in einer der praktischen Übungen der Vorlesung „Programmerzeugungssysteme“ durchgeführt werden könnte. Durch das Feedback der Studenten zur Bedienung und Funktionalität könnte „EBNF-Checker“ evaluiert und mit den bereits angesprochenen Verbesserungen optimiert werden.

5.3 Schlusswort

Ein großer Dank geht an Herrn Prof. Dr.-Ing. Pawelczak für die Betreuung und Unterstützung während der Projekt- und Bachelorarbeit, meine Familie und Nicole.

Durch die Entwicklung der Anwendung und intensive Beschäftigung mit den Grundlagen und Techniken des Compilerbaus und der EBNF konnten die Kenntnisse des bisherigen Studiums vertieft und erweitert werden. Deshalb stellt diese Arbeit als erstes selbständig entwickeltes Softwareprojekt einen gelungenen Abschluss des Bachelorstudiums dar, in der Hoffnung die Weiterbildung der zukünftigen Programmierer mit dieser Lernsoftware fördern zu können und somit der Verpflichtung nach Wirth [1] gerecht zu werden.

6 Anhang

6.1 Quellcode

6.1.1 lexer.jflex

```
1  package lexpase;
2
3  import java_cup.runtime.Symbol;
4  import java_cup.runtime.ComplexSymbolFactory;
5  import java_cup.runtime.ComplexSymbolFactory.Location;
6  import java.io.StringReader;
7
8  %%
9
10 %public
11 %class Scanner
12 %line
13 %column
14 %char
15
16 %cup
17
18 %{
19     private ComplexSymbolFactory symbolFactory; //Verwalten des Symbol-
20     stroms
21     private String scannerAusgabe = ""; //Fehlermeldungen des Scanners
22
23     public Scanner(StringReader in, ComplexSymbolFactory sf){
24         this(in);
25         symbolFactory = sf;
26     }
27
28     //Speichern der Position des Symbols in ComplexSymbolFactory von CUP
29     private Symbol symbol(String name, int sym) {
30         Location left = new Location(yyline+1,yycolumn+1,yychar);
31         Location right = new Location(yyline+1,yycolumn+yylength(),yy-
32 char+yylength());
33         return symbolFactory.newSymbol(name, sym, left, right);
34     }
35
36     //Speichern der Position des Symbols in ComplexSymbolFactory von CUP
37     mit Wert (nur bei Terminal und Nichtterminal)
38     private Symbol symbol(String name, int sym, String val) {
39         Location left = new Location(yyline+1,yycolumn+1,yychar);
40         Location right= new Location(yyline+1,yycolumn+yylength(), yy-
41 char+yylength());
42         return symbolFactory.newSymbol(name, sym, left, right,val);
43     }
44
45     //lexikalischer Fehler (Eingabe unerlaubter Zeichen)
46     private void error(String message) {
47         if(scannerAusgabe.equals("")) {
48             scannerAusgabe = message;
```

```

49         } else if(!message.equals("")) {
50             scannerAusgabe = scannerAusgabe + "\n" + message;
51         }
52     }
53
54     public String getScannerAusgabe() {
55         return scannerAusgabe;
56     }
57
58 %}
59
60 %eofval{
61     Location left = new Location(yyline+1,yycolumn+1,yychar);
62     Location right = new Location(yyline+1,yycolumn+1,yychar+1);
63     return symbolFactory.newSymbol("EOF", ParserSym.EOF, left, right);
64 %eofval}
65
66 %%
67
68 [ \t\r\f\n]+                { break; }
69 "="                          { return symbol("ZUWEISUNG \"]="",
70 ParserSym.ZUWEISUNG); }
71 "."                          { return symbol("ENDE \".\"", Parser-
72 Sym.ENDE); }
73 "("                          { return symbol("KLAMMER_AUF \"(\"", Par-
74 serSym.KLAMMER_AUF); }
75 ")"                          { return symbol("KLAMMER_ZU \")\"", Parser-
76 Sym.KLAMMER_ZU); }
77 "|"                          { return symbol("ODER \"|\"", ParserSym.O-
78 DER); }
79 "["                          { return symbol("OPTION_AUF \"[\"", Parser-
80 Sym.OPTION_AUF); }
81 "]"                          { return symbol("OPTION_ZU \"]\"", Parser-
82 Sym.OPTION_ZU); }
83 "{"                          { return symbol("WDH_AUF \"{\"", Parser-
84 Sym.WDH_AUF); }
85 "}"                          { return symbol("WDH_ZU \"}\"", Parser-
86 Sym.WDH_ZU); }
87
88 "\""(([^\\"]+)|\\") "\""    { return symbol("TERMINAL
89 \"+yytext()+\"", ParserSym.TERMINAL, new String(yytext().substring(1,
90 yytext().length()-1)); }
91 [a-zA-Z][a-zA-Z0-9]*        { return symbol("NICHTTERMINAL
92 \"+yytext()+\"", ParserSym.NICHTTERMINAL, new String(yytext())); }
93
94 .                            { error("Error at line "+(yyline+1)+"", col-
95 umn "+(yycolumn+1)+"": Illegal character \"+yytext()+\" (ignored)"); }

```

6.1.2 parser.cup

```

1  package lexparse;
2
3  import java_cup.runtime.ComplexSymbolFactory;
4  import java_cup.runtime.Symbol;
5  import java.util.HashMap;
6  import java.util.List;
7  import java.util.LinkedList;
8  import tree.*;
9
10 class Parser;
11
12 parser code {:
13
14     private HashMap<String, BinaryTree> treeMap = new HashMap<String, Bina-
15 ryTree>(); //Syntaxbäume der korrekt geparsten Regeln
16     private String parserAusgabe = ""; //Fehlerausgabe des Parsers
17     private boolean syntaxError = false; //Anzeigen, dass ein SyntaxFehler
18 aufgetreten ist
19     private ComplexSymbolFactory.ComplexSymbol aktuelleRegel;
20
21     public Parser(Scanner lex, ComplexSymbolFactory sf) {
22         super(lex,sf);
23     }
24
25     @Override
26     public void report_fatal_error(String message, Object info) {
27         report_error(message, info);
28     }
29
30     //Fehlerbehandlung mit Zeilen- und Spaltenausgabe
31     @Override
32     public void report_error(String message, Object info) {
33         syntaxError = true;
34         String m = "Error";
35         if (info instanceof ComplexSymbolFactory.ComplexSymbol) {
36             ComplexSymbolFactory.ComplexSymbol s = ((ComplexSymbolFac-
37 tory.ComplexSymbol) info);
38             m += " for input symbol " + s.getName() + " in line " +
39 s.xleft.getLine() + ", column " + s.xright.getColumn() + ": " + message;
40         } else {
41             m += ": " + message;
42         }
43         ausgabe(m);
44     }
45
46     //Anzeige der erwarteten Symbole nach Fehler (nicht kompatibel mit GWT
47 2.5)
48     /*@Override
49     protected void report_expected_token_ids(){
50         List<Integer> ids = expected_token_ids();
51         LinkedList<String> list = new LinkedList<String>();
52         for (Integer expected : ids){
53             list.add(symbol_name_from_id(expected));
54         }
55         //System.out.println("instead expected token classes are "+list);
56         ausgabe("instead expected token classes are "+list);
57     }*/
58
59     private void ausgabe(String s) {

```



```

122         if(treeMap.containsKey(tree.getRoot().getValue())) {
123             //Regel schon vorhanden => Fehler
124             report_error("This rule is already defined", aktuel-
125 leRegel);
126         } else {
127             //Regel noch nicht vorhanden => Hinzufügen
128             treeMap.put(tree.getRoot().getValue(), tree);
129         }
130         if(tree.erkenneLinksRekursion(tree.getRoot())){
131             report_error("This rule is left-recursive (interpreting
132 not supported)",aktuelleRegel);
133         }
134     :}
135     | production:p
136     {:
137         BinaryTree tree = new BinaryTree(p);
138         //Optimierungen am Syntaxbaum
139         tree.optimiereAlternative(tree.getRoot());
140         tree.optimiereReihe(tree.getRoot());
141         tree.entferneKlammern(tree.getRoot(), treeMap);
142         tree ersetzeMittigeRekursion(tree.getRoot());
143         tree.optimiereAlternative(tree.getRoot());
144         tree.optimiereReihe(tree.getRoot());
145         if(treeMap.containsKey(tree.getRoot().getValue())) {
146             //Regel schon vorhanden => Fehler
147             report_error("This rule is already defined", aktuel-
148 leRegel);
149         } else {
150             //Regel noch nicht vorhanden => Hinzufügen
151             treeMap.put(tree.getRoot().getValue(), tree);
152         }
153         if(tree.erkenneLinksRekursion(tree.getRoot())){
154             report_error("This rule is left-recursive (interpreting
155 not supported)",aktuelleRegel);
156         }
157     :}
158     ;
159
160 production ::= NICHTTERMINAL:n ZUWEISUNG:z alternative:a ENDE:e
161     {:
162         aktuelleRegel = ((java_cup.runtime.ComplexSymbolFac-
163 tory.ComplexSymbol) CUP$Parser$stack.elementAt(CUP$Parser$top-3));
164         Node temp = new Node(Type.P, n, null, a, null);
165         Node end = new Node(Type.E, null, temp, null, null);
166         temp.setRight(end);
167         a.setTop(temp);
168         RESULT = temp;
169     :}
170     ;
171
172 alternative ::= alternative:a ODER:o reihe:r
173     {:
174         Node temp = new Node(Type.ALTERNATIVE, null, null, r,
175 null);
176         r.setTop(temp);
177         //bei letztem Knoten rechts einfuegen
178         Node zeiger = a;
179         while(zeiger.getRight() != null) {
180             zeiger = zeiger.getRight();
181         }
182         zeiger.setRight(temp);
183         temp.setTop(zeiger);

```

```

184         RESULT = a;
185     :}
186     | reihe:r
187     {:
188         Node temp = new Node(Type.ALTERNATIVE, null, null, r,
189 null);
190         r.setTop(temp);
191         RESULT = temp;
192     :}
193     ;
194
195 reihe ::= reihe:r factor:f
196     {:
197         Node temp = new Node(Type.REIHE, null, null, f, null);
198         f.setTop(temp);
199         //bei letztem Knoten rechts einfuegen
200         Node zeiger = r;
201         while(zeiger.getRight() != null) {
202             zeiger = zeiger.getRight();
203         }
204         zeiger.setRight(temp);
205         temp.setTop(zeiger);
206         RESULT = r;
207     :}
208     | factor:f
209     {:
210         Node temp = new Node(Type.REIHE, null, null, f, null);
211         f.setTop(temp);
212         RESULT = temp;
213     :}
214     ;
215
216 klammer ::= KLAMMER_AUF:k alternative:a KLAMMER_ZU:z
217     {:
218         Node temp = new Node(Type.KLAMMER, null, null, a, null);
219         a.setTop(temp);
220         RESULT = temp;
221     :}
222     ;
223
224 option ::= OPTION_AUF:o alternative:a OPTION_ZU:z
225     {:
226         Node temp = new Node(Type.OPTION, null, null, a, null);
227         a.setTop(temp);
228         RESULT = temp;
229     :}
230     ;
231
232 wiederholung ::= WDH_AUF:w alternative:a WDH_ZU:z
233     {:
234         Node temp = new Node(Type.WDH, null, null, a,
235 null);
236         a.setTop(temp);
237         RESULT = temp;
238     :}
239     ;
240
241 factor ::= NICHTTERMINAL:n
242     {:
243         RESULT = new Node(Type.NT, n, null, null, null);
244     :}
245     |

```



```
246         TERMINAL:t
247         {:
248             RESULT = new Node(Type.T, t, null, null, null);
249         :}
250         |
251         klammer:k
252         {:
253             RESULT = k;
254         :}
255         |
256         option:o
257         {:
258             RESULT = o;
259         :}
260         |
261         wiederholung:w
262         {:
263             RESULT = w;
264         :}
265         ;
```

6.1.3 Kommentare zu Klassendiagramm Interpreter.java

```
1  /*Parser-Instanz (CUP)*/
2  private Parser myParser;
3  /*Scanner-Instanz (JFlex)*/
4  private Scanner myScanner;
5  /*eingegebene EBNF-Grammatik*/
6  private String grammatik;
7  /*Eingabe zur EBNF-Grammatik*/
8  private String eingabe;
9  /*HashMap mit allen ASTs*/
10 private HashMap<String, BinaryTree> treeMap;
11 /*Fehlerausgabe des Scanners und Parsers*/
12 private String lexParseAusgabe;
13 /*Fehlerausgabe des Interpreters*/
14 private String interpreterAusgabe = "";
15 /*Fehlerkette der Regeln*/
16 private String fehlerAusgabe = "";
17 /*zeigt Syntaxfehler an*/
18 private boolean syntaxError = false;
19 /*zeigt Fehler beim Interpretieren an*/
20 private boolean interpretError = false;
21 /*zeigt an, dass eine Regel nicht gefunden wurde*/
22 private boolean ruleNotFound = false;
23 /*interne Darstellung der "eingabe"*/
24 private Stack<String> textStapel;
25
26 /*Erzeugen einer neuen Interpreter-Instanz*/
27 public void Interpreter();
28 /*Scannen und Parsen der "grammatik"*/
29 public void initParsing();
30 /*Interpretieren der "eingabe" zur "grammatik"*/
31 public void initInterpreting(String startregel);
32 /*wird von "initInterpreting" zum Interpretieren verwendet*/
33 public void interpret(Node regel);
34
35 /*Notwendige Getter- und Settermethoden*/
36 public void setGrammatik(String grammatik);
37 public void setEingabe(String eingabe);
38 public HashMap<String, BinaryTree> getTreeMap();
39 public String getLexParseAusgabe();
40 public String getInterpreterAusgabe();
41 public boolean isSyntaxError();
42 public boolean isInterpretError();
```

6.1.4 ListElement.java

```
1  /*
2   * To change this license header, choose License Headers in Project Proper-
3   * ties.
4   * To change this template file, choose Tools | Templates
5   * and open the template in the editor.
6   */
7  package interpreter;
8
9  import java.util.Stack;
10
11  /**
12   *
13   * @author Admin
14   */
15  public class ListElement {
16      private int tiefe; //Tiefe
17      private Stack<String> werte = new Stack<String>(); //genommene Zeichen
18      bei entsprechender Tiefe
19
20      public ListElement(int tiefe, String value) {
21          this.tiefe = tiefe;
22          werte.push(value);
23      }
24
25      public int getTiefe() {
26          return tiefe;
27      }
28
29      public void setTiefe(int tiefe) {
30          this.tiefe = tiefe;
31      }
32
33      public Stack<String> getWerte() {
34          return werte;
35      }
36
37
38  }
```

6.1.5 Resources.java

```
1  package org.ebnfChecker.gui.client;
2
3  import com.google.gwt.core.client.GWT;
4  import com.google.gwt.resources.client.ClientBundle;
5  import com.google.gwt.resources.client.CssResource;
6
7  public interface Resources extends ClientBundle {
8
9      public static final Resources INSTANCE = GWT.create(Resources.class);
10     @Source("myStylesheet.css")
11     @CssResource.NotStrict
12     CssResource css();
13 }
```

6.2 Verwendete Software

- *NetBeans IDE 8.0.2* (Entwicklungsumgebung), <https://netbeans.org/>,
[Zugriff am: 16.06.2015]
 - *JDK 1.8* (Java Development Kit),
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>,
[Zugriff am: 18.06.2015]
 - *GWT4NB 2.10.12* („NetBeans-Plugin für GWT“),
<https://github.com/ksfreitas/gwt4nb>, [Zugriff am: 18.06.2015]
 - *GlassFish Server 3*, NetBeans-Pluginmanager (Version 1.63.1)
- *JFlex 1.6.0* (Scanner-Generator), <http://sourceforge.net/projects/jflex/files/jflex/1.6.0/>,
[Zugriff am: 16.06.2015]
- *Java CUP 11b 2014-12-04* (Parser-Generator),
<http://www2.cs.tum.edu/projects/cup/releases/>, [Zugriff am: 16.06.2015]
- *GWT 2.5.0* (Google Web Toolkit), <http://www.gwtproject.org/versions.html>,
[Zugriff am: 16.06.2015]
Das „*GWT4NB*“-Plugin unterstützt keine neuere Ausführung der GWT-SDK.
- *Ahome-Client-IO 2.0.0*, <https://github.com/ahome-it/ahome-client-io>,
[Zugriff am: 16.06.2015]
- *Enterprise Architect 10* (UML-Diagramme), <http://www.sparxsystems.de/>,
[Zugriff am: 16.06.2015]
- *Snipping Tool* (Screenshots)
- *Microsoft Power Point 2013* (Abbildungen)
- *Microsoft Word 2013* (Bachelorarbeit)

6.3 Gliederung der beigelegten DVD

- Elektronische Version der Bachelorarbeit im PDF-Format
- NetBeans-Projektordner
 - „Simulator“ („EBNF-Simulator“ ohne GWT-Weboberfläche)
 - „EBNF_Checker“ (vollständiges Softwareprojekt)
- Quellen
 - „*JFlex User's Manual*“ [17,19]
 - „*video2brain – GWT – Crashkurs*“ [30,33]

Abbildungsverzeichnis

ABBILDUNG 2.1: SELBSTDEFINITION DER EBNF [7]	3
ABBILDUNG 2.2: KERNELEMENTE DER EBNF [12]	4
ABBILDUNG 2.3: STRUKTUR DER JFLEX-DATEI [19]	5
ABBILDUNG 3.1: LOGO „EBNF-CHECKER“	9
ABBILDUNG 3.2: ÜBERSICHT ÜBER DIE ANWENDUNGSFÄLLE DES PROGRAMMS „EBNF-CHECKER“	10
ABBILDUNG 3.3: NOTWENDIGE EINGABEN DES BENUTZERS	11
ABBILDUNG 3.4: ERZEUGEN DES SCANNERS MIT JFLEX	11
ABBILDUNG 3.5: ERZEUGEN DES PARSERS MIT CUP	12
ABBILDUNG 3.6: SYSTEMÜBERSICHT „EBNF-SIMULATOR“	12
ABBILDUNG 3.7: ANGEPASSTE SELBSTDEFINITION DER EBNF	13
ABBILDUNG 3.8: ENDGÜLTIG ANGEPASSTE SELBSTDEFINITION DER EBNF	15
ABBILDUNG 3.9: KLAMMERPRIORITÄTEN IN PARSE.CUP [ANHANG 6.1.2, Z. 95-98]	16
ABBILDUNG 3.10: PRIORITÄTEN DER EBNF-KLAMMERARTEN	16
ABBILDUNG 3.11: ANGEPASSTE EBNF-SELBSTDEFINITION [ABBILDUNG 3.8] IN BNF ZUR VERWENDUNG IN JAVA-CUP	16
ABBILDUNG 3.12: KLASSENDIAGRAMME DER KLASSEN „NODE“, „TYPE“ UND „BINARYTREE“ (SYNTAXBAUMERZEUGUNG)	17
ABBILDUNG 3.13: PRINZIPIELLER AUFBAU DER SYNTAXBÄUME	18
ABBILDUNG 3.14: WIEDERHOLUNGEN UND OPTIONEN IM SYNTAXBAUM	18
ABBILDUNG 3.15: SYNTAXBAUMERZEUGUNG DER REGEL „WIEDERHOLUNG“ IN PARSE.CUP [ANHANG 6.1.2, Z. 232 - 239]	19
ABBILDUNG 3.16: GRAMMATIKBEISPIEL ZUR SYNTAXBAUMERZEUGUNG	19
ABBILDUNG 3.17: OPTIMIERTE SYNTAXBÄUME DES GRAMMATIKBEISPIELS [ABBILDUNG 3.16]	19
ABBILDUNG 3.18: PARSE-BAUM DER REGEL „DIGIT“ [ABBILDUNG 3.16] VOR OPTIMIERUNG	20
ABBILDUNG 3.19: BEISPIEL FÜR FEHLERHAFTES GRAMMATIKBEISPIEL	21
ABBILDUNG 3.20: ERZEUGTE FEHLERMELDUNGEN BEI EINGABE DES FEHLERHAFTEN GRAMMATIKBEISPIELS [ABBILDUNG 3.19]	21
ABBILDUNG 3.21: FEHLERMELDUNG BEI LINKSREKURSIVEN GRAMMATIKBEISPIELN	22
ABBILDUNG 3.22: ERFOLGREICHES PARSEN OHNE FEHLER	22
ABBILDUNG 3.23: KLASSENDIAGRAMM „INTERPRETER“	23
ABBILDUNG 3.24: TYPUNTERSCHIED MIT SWITCH-CASE-KONSTRUKT	24
ABBILDUNG 3.25: AKTIVITÄTSDIAGRAMM DER REGEL TERMINAL („T“)	25
ABBILDUNG 3.26: AKTIVITÄTSDIAGRAMM DER REGEL NICHTTERMINAL („NT“)	26
ABBILDUNG 3.27: AKTIVITÄTSDIAGRAMM DER REGEL REIHE („REIHE“)	27
ABBILDUNG 3.28: AKTIVITÄTSDIAGRAMM DER REGEL ALTERNATIVE („ALTERNATIVE“)	28
ABBILDUNG 3.29: AKTIVITÄTSDIAGRAMM DER REGEL OPTION („OPTION“)	28
ABBILDUNG 3.30: GRAMMATIKBEISPIEL FÜR ZURÜCKLEGEN BEI OPTIONEN	29
ABBILDUNG 3.31: BEISPIEL FÜR KOMBINATIONSMÖGLICHKEITEN DER OPTIONEN	29
ABBILDUNG 3.32: AKTIVITÄTSDIAGRAMM DER REGEL WIEDERHOLUNG („WDH“)	30
ABBILDUNG 3.33: BEISPIEL-GRAMMATIK FÜR „WDH“	30

ABBILDUNG 3.34: AKTIVITÄTSDIAGRAMM DER REGEL REIHE FÜR DEN LINKEN NACHFOLGERTYP „WDH“	31
ABBILDUNG 3.35: BEISPIEL FÜR RECHTSSEITIGE REKURSION	32
ABBILDUNG 3.36: INTERPRETATION DER EINGABE „2015“ MIT REKURSIVER GRAMMATIKREGEL „NUMBER“ [ABBILDUNG 3.35]	32
ABBILDUNG 3.37: UMWANDLUNG VON MITTIG-REKURSIVEN REGELN	33
ABBILDUNG 3.38: SYNTAXBAUM EINER MITTIGE-REKURSIVEN REGEL VOR UMWANDLUNG.....	33
ABBILDUNG 3.39: SYNTAXBAUM EINER UMGEWANDELTEN MITTIG-REKURSIVEN REGEL	34
ABBILDUNG 3.40: BEISPIEL FÜR MITTIG-REKURSIVE GRAMMATIKREGELN	34
ABBILDUNG 3.41: FEHLERMELDUNGEN BEI NICHT DEKLARIERTEN REGELN	35
ABBILDUNG 3.42: ABBRUCH DER REKURSION BEI MEHR ALS 500 AUFRUFEN IN DER KLASSE „INTERPRETER“ ...	35
ABBILDUNG 3.43: FEHLERMELDUNGEN EINES ÜBER ZWEI EBENEN LINKSREKURSIVEN GRAMMATIKBEISPIELS .	36
ABBILDUNG 3.44: FEHLERMELDUNG EINER FALSCHEN EINGABE ZU EINER KORREKTEN BEISPIELGRAMMATIK .	36
ABBILDUNG 3.45: KORREKTES INTERPRETIEREN OHNE SEMANTISCHE FEHLER	36
ABBILDUNG 3.46: „TEXTAREA-WIDGET“	38
ABBILDUNG 3.47: PARSEN EINER FEHLERHAFTEN GRAMMATIK MIT ENTSPRECHENDEN MELDUNGEN	38
ABBILDUNG 3.48: BEISPIEL FÜR SYNTAXBAUMAUSGABE	38
ABBILDUNG 3.49: ERZEUGEN DES ATTRIBUTS „ROOTPANEL“	39
ABBILDUNG 3.50: INITIALISIERUNG CLIENTIO.....	39
ABBILDUNG 3.51: SPEICHERN DER EBNF-GRAMMATIK IN TEXTDATEI	40
ABBILDUNG 3.52: HOCHLADEN EINER EBNF-GRAMMATIK AUS EINER TEXTDATEI	40
ABBILDUNG 3.53: DARSTELLUNG DES SPEICHERVORGANGS MIT CLIENTIO AUF DER WEBOBERFLÄCHE	41
ABBILDUNG 3.54: ANLEGEN EINES BUTTON-WIDGETS UND ZUWEISEN EINER EINDEUTIGEN ID	41
ABBILDUNG 3.55: ANPASSEN DES DESIGNS VON GWT-KLASSEN UND GWT-OBJEKTEN MIT CSS.....	41
ABBILDUNG 3.56: EINBINDEN DER CSS-DATEI „MYSTYLESHEET.CSS“	42
ABBILDUNG 3.57: NICHT UNTERSTÜTZTE IMPORTE DES „EBNF-SIMULATORS“	42
ABBILDUNG 3.58: „SUPER-SOURCE“	43
ABBILDUNG 4.1: BEISPIEL 1 FÜR GRAMMATIKEINGABE: EBNF-SELBSTDEFINITION [7]	45
ABBILDUNG 4.2: EINGABEN ZU BEISPIEL 1 FÜR GRAMMATIKEINGABE [ABBILDUNG 4.1]	46
ABBILDUNG 4.3: BEISPIEL 2 FÜR GRAMMATIKEINGABE: INTEGER-LITERALE IN JAVA [38]	46
ABBILDUNG 4.4: EINGABEN ZU BEISPIEL 2 FÜR GRAMMATIKEINGABE [ABBILDUNG 4.3]	46
ABBILDUNG 4.5: BEISPIEL 3 FÜR GRAMMATIKEINGABE: MINIB [39].....	47
ABBILDUNG 4.6: EINGABEN ZU BEISPIEL 3 FÜR GRAMMATIKEINGABE [ABBILDUNG 4.5]	48
ABBILDUNG 4.7: STARTOBERFLÄCHE „EBNF-CHECKER“	48
ABBILDUNG 4.8: POPUP-OBERFLÄCHE NACH KLICKEN AUF „ABOUT“ (HILFE ANZEIGEN LASSEN)	49
ABBILDUNG 4.9: WEBOBERFLÄCHE NACH PARSEN UND INTERPRETIEREN DES BEISPIELS AUS „SHOW EXAMPLE“	50

Literaturverzeichnis

- [1] *A Few Words with Niklaus Wirth (Carlo Pescio)*, http://www.eptacom.net/pubblicazioni/pub_eng/wirth.html,
[Zugriff am: 17.06.2015]
- [2] N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, Bonn, 1996, Kap. 1
- [3] N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, Bonn, 1996, S. 7
- [4] *ISO/IEC 14977:1996*, http://iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=26153,
[Zugriff am: 16.06.2015]
- [5] N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, Bonn, 1996, Kap. 2
- [6] Dieter Pawelczak: *Programmerzeugungssysteme*, Kurzsriptum zur Lehrveranstaltung WT2015, Fakultät ETTI, Universität der Bundeswehr München, 2015, Kap. 2.2
- [7] N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, Bonn, 1996, S. 8
- [8] D. Pawelczak: *Start in die Technische Informatik*, Shaker Verlag, Aachen, S.68
- [9] D. Pawelczak: *Start in die Technische Informatik*, Shaker Verlag, Aachen, S. 42
- [10] *Java Escape Sequences*, <https://docs.oracle.com/javase/tutorial/java/data/characters.html>,
[Zugriff am: 20.06.2015]
- [11] D. Pawelczak: *Start in die Technische Informatik*, Shaker Verlag, Aachen, Kap. 2 – Formale Sprachen
- [12] Dieter Pawelczak: *Programmerzeugungssysteme*, Kurzsriptum zur Lehrveranstaltung WT2015, Fakultät ETTI, Universität der Bundeswehr München, 2015, S.11 – Bild 2.4
- [13] Dieter Pawelczak: *Programmerzeugungssysteme*, Kurzsriptum zur Lehrveranstaltung WT2015, Fakultät ETTI, Universität der Bundeswehr München, 2015, Kap. 3.1 - 3.2
- [14] *The Lex & Yacc Page*, <http://www.dinosaur.compilertools.net/>, [Zugriff am: 16.06.2015]
- [15] *flex: The Fast Lexical Analyzer*, <http://www.flex.sourceforge.net/>, [Zugriff am: 16.06.2015]
- [16] A. Aho, M. Lam, R. Sethi, J. Ullman, *Compiler – Prinzipien, Techniken und Werkzeuge*, Pearson Studium, München, 2. Auflage, 2008, Kap. 3.6 (S. 178ff.)
- [17] Gerwin Klein, Steve Rowe, Régis Décamps: *JFlex User's Manual*, Version 1.6, 20.06.2014, Kap. 3.3
- [18] *JFlex – The Fast Scanner Generator for Java*, <http://www.jflex.de/>, [Zugriff am: 16.06.2015].
- [19] Gerwin Klein, Steve Rowe, Régis Décamps: *JFlex User's Manual*, Version 1.6, 20.06.2014
- [20] Dieter Pawelczak: *Programmerzeugungssysteme*, Kurzsriptum zur Lehrveranstaltung WT2015, Fakultät ETTI, Universität der Bundeswehr München, 2015, Kap. 4.1 – 4.2
- [21] N. Wirth: *Grundlagen und Techniken des Compilerbaus*, Addison-Wesley, Bonn, 1996, Kap. 4.3
- [22] A. Aho, M. Lam, R. Sethi, J. Ullman, *Compiler – Prinzipien, Techniken und Werkzeuge*, Pearson Studium, München, 2. Auflage, 2008, Kap. 4.7.4 (S. 319ff.)
- [23] *Bison – GNU Project*, <http://www.gnu.org/software/bison/>, [Zugriff am: 16.06.2015]
- [24] Dieter Pawelczak: *Programmerzeugungssysteme*, Kurzsriptum zur Lehrveranstaltung WT2015, Fakultät ETTI, Universität der Bundeswehr München, 2015, S. 72
- [25] *Cup – LALR Parser for Java*, <http://www2.cs.tum.edu/projects/cup/index.php>, [Zugriff am: 16.06.2015].
- [26] H. Schulte: *Fachverzeichnis – Informationstechnologie von A-Z – Abkürzungen*, WEKA, Kissing, Oktober 2007, S. 1093 (WORA)
- [27] *GWT – JRE Emulation*, <http://www.gwtproject.org/doc/latest/RefJreEmulation.html>,
[Zugriff am: 16.06.2015]

-
- [28] H. Schulte: *Fachverzeichnis – Informationstechnologie von A-Z – Abkürzungen*, WEKA, Kissing, Oktober 2007, S. 38 (AJAX)
- [29] GWT, <http://www.gwtproject.org/>, [Zugriff am: 16.06.2015]
- [30] Steyer Ralph, *video2brain - GWT – Crashkurs*, <https://www.video2brain.com/de/videotraining/gwt-crashkurs>, [Zugriff am: 16.06.2015], Kap. 1.1 – 1.2
- [31] Andrea Baumann: *Software Engineering – 07 Muster im Software Engineering*, Vorlesungsfolien FT2014, Fakultät ETTI, Universität der Bundeswehr München, 2014, S. 4-5
- [32] A. Aho, M. Lam, R. Sethi, J. Ullman, *Compiler – Prinzipien, Techniken und Werkzeuge*, Pearson Studium, München, 2. Auflage, 2008, Kap. 4.2.4 (S. 244ff.)
- [33] Steyer Ralph, *video2brain - GWT – Crashkurs*, <https://www.video2brain.com/de/videotraining/gwt-crashkurs>, [Zugriff am: 16.06.2015], Kap. 2.1
- [34] GWT – *Widget Gallery*, <http://www.gwtproject.org/doc/latest/RefWidgetGallery.html>, [Zugriff am: 16.06.2015]
- [35] *Ahome-Client-IO*, <https://github.com/ahome-it/ahome-client-io>, [Zugriff am 16.06.2015]
- [36] GWT – *Overriding one package implementation with another*, <http://www.gwtproject.org/doc/latest/DevGuideOrganizingProjects.html>, [Zugriff am: 16.06.2015]
- [37] GWT Emulationen für „StringReader“ und „Reader“ aus „java.io“, <https://code.google.com/p/anhquan/source/browse/trunk/Quiz/docs/restlet-gwt/org.restlet.com/google/gwt/emul/java/io/?r=60>, [Zugriff am 16.06.2015]
- [38] *Karlsruher Institut für Technologie - EBNF*, <http://formal.iti.kit.edu/beckert/teaching/InformatikIM-WS0506/iim06.pdf>, S. 5, [Zugriff am: 17.06.2015]
- [39] Dieter Pawelczak: *Programmerzeugungssysteme*, Kurzsriptum zur Lehrveranstaltung WT2015, Fakultät ETTI, Universität der Bundeswehr München, 2015, S.15
- [40] Dieter Pawelczak: *Programmerzeugungssysteme*, Kurzsriptum zur Lehrveranstaltung WT2015, Fakultät ETTI, Universität der Bundeswehr München, 2015, S.16
- [41] GWT - *HTML5 Storage*, <http://www.gwtproject.org/doc/latest/DevGuideHtml5Storage.html>, [Zugriff am: 18.06.2015]
- [42] *Interface JField*, <http://www.gwtproject.org/javadoc/latest/com/google/gwt/core/ext/typeinfo/JField.html>, [Zugriff am: 18.06.2016]
- [43] *Virtual-C IDE*, <https://sites.google.com/site/virtualcide/>, [Zugriff am: 18.06.2015]