Fachhochschule Nordostniedersachsen
Fachbereich Wirtschaft

Konzeption und Implementierung eines Prüfwerkzeuges für den Dimensional Measuring Interface Standard

DIPLOMARBEIT
zur Erlangung des akademischen Grades
Diplom – Wirtschaftsinformatiker (FH)
März 2004

Erstgutachter: Prof. Dr. rer. nat. Jürgen Jacobs
Zweitgutachter: Dr. Frank Röhrdanz (Volkswagen AG)

Vorgelegt von: Jan Reimers
Triftstraße 100
21075 Hamburg
Matrikel-Nr. 152 279
7. Semester

INHALTSVERZEICHNIS

DARSTELLUNGSVERZEICHNIS

ABKÜRZUNGSVERZEICHNIS

ANSI	American National Standards Institute
ANTLR	Another Tool For Language Recognition
AST	Abstract Syntax Tree
CAD	Computer Aided Design
CAM-I	Consortium For Advanced Manufacturing – International
DIN	Deutsches Institut für Normung
DMI	DMIS-Input-Format
DMIS	Dimensional Measuring Interface Standard
DMO	DMIS-Output-Format
DTD	Document Type Definition
EBNF	Extended Backus-Naur-Form
ISO	International Organization for Standardization
JavaCC	Java Compiler Compiler
KMG	Koordinaten-Mess-Gerät
lk	Lexer- / Scanner-Lookahead
pk	Parser-Lookahead
QS-INFO	Qualitätssicherungs-Information
QUIRL	Qualität im Rohbau Leitstand
TERAMESS	Terminplanungs- und Auftragsabwicklungssystem für Messräume
XML	Extensible Markup Language

VORWORT

Die vorliegende Diplomarbeit entstand im Rahmen meines Praxissemesters bei der VOLKSWAGEN AG in Wolfsburg.

Die Diplomarbeit enthält somit den Praxisbericht.

Für die Unterstützung danke ich meinem Betreuer Stefan Prange und allen anderen Kollegen, insbesondere Lars Koch für den gemeinsamen Besuch in einem Messraum, für Echtdaten zum Testen und für das Auswerten der Fehlerberichte und Thomas Riecher für Anregungen und Auskünfte. Ferner gebührt all denen Dank, die an der Entwicklung der Open-Source-Projekte *Eclipse*, *ANTLR* und *OpenOffice* mitgewirkt haben.

Hamburg, den 06. März 2004

A Motivation

In der Abteilung „Informationstechnik Elektronik- und Qualitätsinformation – Messen und Prüfen“ wird für den gesamten Volkswagen Konzern Software erstellt, die unter anderem in der Qualitätssicherung eingesetzt wird. Die von den Anwendern zur Datenerhebung benutzten Koordinaten-Mess-Geräte werden über Programm-Dateien gesteuert und erzeugen während der Ausführung eines solchen Programms Dateien, in denen der Messvorgang protokolliert ist. Ein- und Ausgabe-Dateiformat sind durch den Dimensional Measuring Interface Standard (DMIS, bei Volkswagen als „deh-miss“ ausgesprochen) festgelegt.

Ein Standard ist keine gesetzliche Festlegung, sondern eine Übereinkunft verschiedener betroffener Gruppen. Es ist also niemand verpflichtet, sich an den Standard zu halten. Ein Standard bietet jedoch einige Vorteile. So wäre es zum Beispiel ausreichend, ein Messprogramm einmal zu erstellen, auch wenn es von Messmaschinen verschiedener Hersteller ausgeführt werden soll. Es entsteht also keine Abhängigkeit von einem bestimmten Hersteller. Folglich ermöglicht ein Standard ein wesentlich wirtschaftlicheres Arbeiten, als wenn für die Messmaschinen verschiedener Hersteller jeweils Sonderbehandlungen nötig wären.

Das Vorhandensein eines Standards bedeutet jedoch nicht, dass diesem jede Datei des entsprechenden Formats genügt. Verletzungen des Standards machen jedoch dessen Vorteile zunichte. So müssen womöglich für jeden einzelnen Messmaschinenhersteller einige kleine Anpassungen an der Software zur Datenaufbereitung vorgenommen werden. Hat ein Hersteller mit der Unterstützung von DMIS für sein Produkt geworben, handelt es sich sogar um ein Fehlen von zugesicherten Eigenschaften.

Um eine Prüfung von DMIS-Dateien auf Einhaltung des Standards schnell und verlässlich durchführen zu können, soll das Prüfwerkzeug *DMIS-Checker* entwickelt werden, das lexikalische und syntaktische Fehler findet, indem es die Dateien analysiert. Hierzu ist die Verwendung eines Parser-Generators angedacht, weil angestrebt wird, dass der DMIS-Checker an kommende Versionen des DMIS angepasst werden kann, und zwar allein durch

Änderungen an einer maschinell verarbeitbaren Sprachbeschreibung. Es besteht der Wunsch, dass eine solche Anpassung möglichst wenig Kenntnisse im Compilerbau erfordert.

Da die Beschreibung des DMIS-Formats bereits unter anderem in Form einer Grammatik, also in für einen Rechner verständlicher Form, vorliegt, soll zunächst geprüft werden, ob diese als Eingabe für einen Parser-Generator geeignet ist. Ist das nicht der Fall, muss eine geeignete Grammatik erstellt werden. Dafür könnte ein *Grammatik-Generator* entwickelt werden, der die Grammatik in das benötigte Format übersetzt. Dieser müsste weitgehend selbständig arbeiten, um dem Wunsch nach einfacher Anpassbarkeit gerecht zu werden.

B Projekt-Vorbereitung

Der Anwendungsbereich des Dimensional Measuring Interface Standards ist nur bei bestimmten Industriezweigen zu finden. Entsprechend ist dieser Standard nicht allgemein bekannt. Daher werden vor der eigentlichen Projekt-Durchführung Zweck, Einsatzgebiet und Möglichkeiten erläutert.

Auch wenn die Übersetzung von Quellcode und das Parsen von Dateien in der Informatik alltäglich sind, können Kenntnisse im Compilerbau nicht vorausgesetzt werden. Daher wird eine kurze Einführung in dieses Thema gegeben.

1 Messen und Prüfen in der Qualitätssicherung

Qualitätssicherung bedeutet, Produkte mit ihrem zu Grunde liegenden Modell zu vergleichen. Werden dabei Abweichungen festgestellt, muss der Fertigungsprozess an geeigneter Stelle verändert werden. Um das reale Produkt mit einem Modell vergleichen zu können, müssen Messungen durchgeführt werden. Die so ermittelten Ist-Daten werden jedoch nie genau mit den Soll-Daten des CAD-Modells übereinstimmen, weil einerseits die Messgenauigkeit beschränkt ist, und andererseits die Maschinen in der Fertigung an technische Grenzen stoßen. Präziser arbeitende Maschinen wären zwar technisch machbar, jedoch ökonomisch nicht sinnvoll. Daher werden Toleranzen festgelegt.

1.1 Koordinaten-Mess-Geräte

In dieser Arbeit geht es um räumliche Messungen, für die Koordinaten-Mess-Geräte (KMG) eingesetzt werden. Solche Messgeräte bestimmen die Koordinaten eines Punktes oder eines komplexeren geometrischen Gebildes, Merkmal genannt, im dreidimensionalen Raum.

Quelle: Website der Carl Zeiss Gruppe

Darstellung 1: KMG (Einzel-Ständer)

Darstellung 1 zeigt ein solches Messgerät. Es besteht aus einem senkrechten Turm (Ständer, im Bild weiß mit Aufschrift), einem waagerechten Arm (Pinole, im Bild hellgrau, ungefähr in der Mitte), und einem daran befestigten Sensor – zum Beispiel einem Taster.

Unter Verwendung des üblichen Koordinatensystems ist der Ständer in X-Richtung beweglich, und die Pinole wird in Y-Richtung ein- und aus-, beziehungsweise in Z-Richtung hoch- und heruntergefahren. Die Messgeräte werden von einem Rechner gesteuert. Der Tastkopf ist schwenk- und drehbar, so dass verschiedenste Punkte an einem zu vermessenden Werkstück erreicht werden können. KMGs gibt es in verschiedenen Bauweisen, zum Beispiel Einzel- oder Doppelständergeräte, und in verschiedenen Größen.

Quelle: Website der Carl Zeiss Gruppe

Darstellung 2: Arm mit Taster

In Darstellung 2 ist die Pinole eines KMGs zu sehen, an deren Ende sich ein Taster befindet, der nach unten geschwenkt und leicht gedreht wurde. Die Antastung eines Punktes auf dem Werkstück – hier ein Wasserhahn – steht kurz bevor.

Die Messung erfolgt, indem der Taster in eine bestimmte Position gebracht wird und dann in vorgegebener Richtung – möglichst senkrecht zur Fläche – auf das Werkstück zufährt, bis der Taster Kontakt hat. Im Moment des Kontakts werden Arm- und Tasterstellung festgehalten und an Hand eines

zuvor gesetzten Koordinatenursprungs in die Koordinaten des Punktes umgerechnet.[1] Dabei muss darauf geachtet werden, dass zwischen Taster und zu messendem Punkt auf dem Werkstück keine Hindernisse sind. Andernfalls wird nicht der gewünschte, sondern irgendein Punkt auf dem Hindernis gemessen.

Die Genauigkeit von Koordinaten-Mess-Geräten hängt unter anderem von ihrer Bauart, ihren Korrektur-Algorithmen und den Umgebungsbedingungen wie Temperatur und Luftfeuchtigkeit während der Messung ab. Die Spanne reicht von 1/10 bis 1/1000 Millimeter. Bei Karosserieteilen wird meist im Bereich von 1/100 Millimeter gearbeitet.

Quelle: Website der Carl Zeiss Gruppe

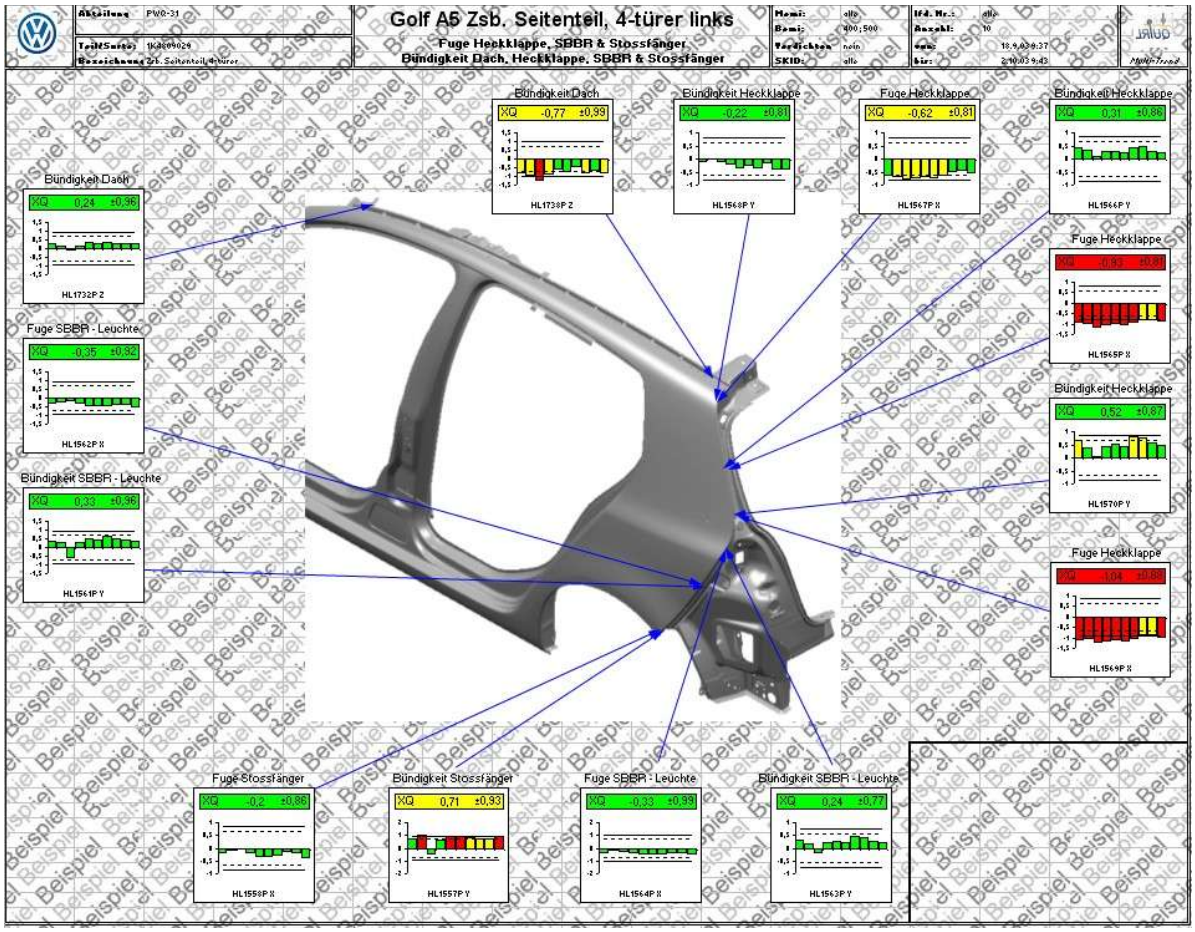
Darstellung 3: Messraum

Darstellung 3 zeigt einen Messraum. Es sind zwei Ständer und die zugehörigen Steuerpulte sowie eine zu vermessende Autotür zu sehen. Die Vermessung eines Werkstücks kann auch mit mehreren (üblich sind zwei) Ständern parallel durchgeführt werden. Dieses Vorgehen bietet sich zum Beispiel an, wenn – wie oben zu sehen – nicht alle Punkte mit einem Arm erreicht werden können, ohne das Werkstück zu bewegen.

Die steigenden Qualitätsanforderungen haben ein erhöhtes Datenaufkommen in der Qualitätssicherung zur Folge. Daher werden die Messdaten zum

1 Vgl. Wocke, P. M., Koordinatenmeßmaschinen, 1993, S. 2

Beispiel durch das Produkt QUIRL analysiert und grafisch aufbereitet und durch QS-INFO verwaltet.



Quelle: VOLKSWAGEN AG

Darstellung 4: Beispiel für ein zu vermessendes Bauteil mit angetragenen Messpunkten

Ein Beispiel für die aufbereiteten Messdaten eines Karosserieteils zeigt Darstellung 4. Die Pfeile verbinden jeweils ein Kästchen, das die ermittelten Abweichungen mehrerer Messungen eines Punktes darstellt, mit dem zugehörigen Punkt. In den Diagrammen sind auch die oben angesprochenen Toleranzen erkennbar, nämlich die durchgehenden waagerechten schwarzen Linien. Grüne Balken stellen zum Beispiel einen Messwert dar, der innerhalb der Toleranz liegt, und rote Balken einen Messwert, der außerhalb der Toleranz liegt.

1.2 Dimensional Measuring Interface Standard

Der Dimensional Measuring Interface Standard wurde von dem American National Standards Institute (ANSI) entwickelt. Laut der Beschreibung des DMIS-Dokumentes im Online-Shop des ANSI[1] ist das Ziel von DMIS die Bereitstellung eines Standards für die bidirektionale Kommunikation zwischen Computersystemen und Prüfgeräten. Dazu stellt DMIS eine Sammlung von Bestimmungen für Prüfprogramme und Prüfergebnisse bereit.[2] Prüfprogramme werden auch als DMIS-Input (DMI) und Prüfergebnisse als DMIS-Output (DMO) bezeichnet.

Aus der Beschreibung der Firma Object Workshops Inc. geht hervor, dass DMIS ursprünglich als unabhängiges Austauschformat gedacht war, inzwischen jedoch zu einer vollwertigen Prüf-Programmiersprache herangewachsen ist. Ferner soll laut dieser Quelle DMIS zu einem ISO-Standard werden.

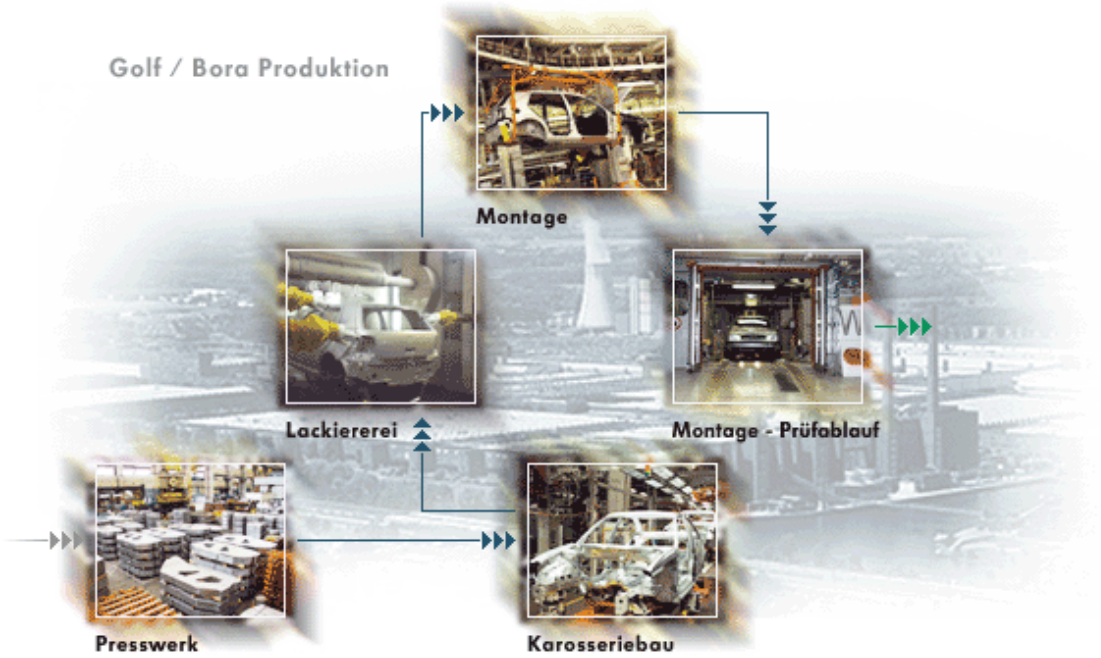
Der Standard DMIS 04.0 besteht aus zwei Teilen. Teil 1 legt DMIS-Input und -Output zur Offline-Kommunikation mit KMGs fest. Teil 2 definiert basierend auf Teil 1 eine objektorientierte Programmierschnittstelle zur Online-Kommunikation mit KMGs und befindet sich in der letzten Stufe der Standardisierung.[3] In dieser Arbeit wird nur Teil 1 berücksichtigt, da der zweite Teil kein Dateiformat beschreibt, dessen Einhaltung überprüft werden könnte.

1 Eine andere Beschreibung gibt es auf der Website der ANSI nicht. Stattdessen wird ein Katalog mit Beschreibungen zu allen verfügbaren ANSI-Standards zum Kauf angeboten.

2 Vgl. ANSI, Document Details, 2003

3 Vgl. Object Workshops, DMIS, 2003

1.2.1 Einordnung in die Wertschöpfungskette



Quelle: In Anlehnung an VOLKSWAGEN AG, Intranet, Besichtigungen Online

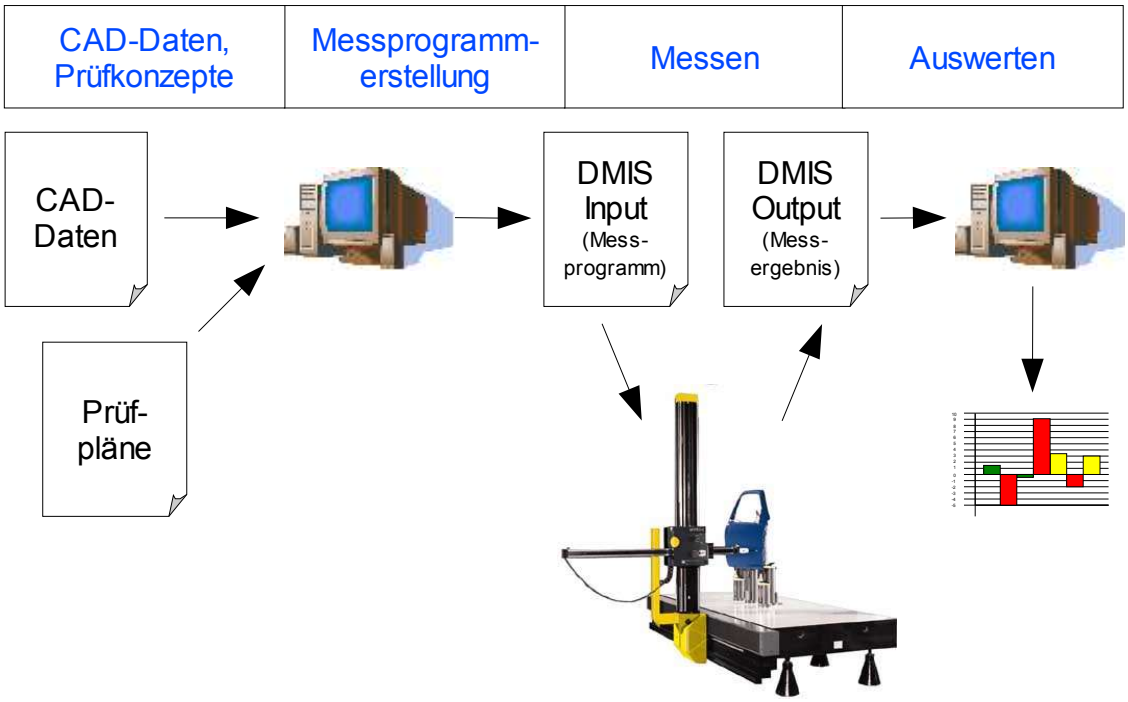
Darstellung 5: Prozesskette am Beispiel Golf / Bora Produktion

Darstellung 5 zeigt die Prozesskette der Automobilfertigung im Werk Wolfsburg am Beispiel der Golf / Bora Produktion.

Als erster Schritt in der Automobilfertigung werden im Presswerk Blech-Formteile für die Karosserie hergestellt. Bereits hier findet unter anderem eine maßliche Überprüfung stichprobenartig statt. Die produzierten Teile werden im Fertigungsabschnitt Karosseriebau zu einer Rohkarosse zusammengefügt. Wiederum werden in Stichproben Qualitätskontrollen durchgeführt. Dabei werden alle Karossenmaße mit den Daten aus dem CAD-Modell verglichen. Während der Montage und der Abschlussprüfung finden keine Prüfungen mit KMGs statt.[1] Neben der Karosseriefertigung werden KMGs in der Aggregate-Fertigung eingesetzt.

Zur Prüfung eines bestimmten Bauteils oder einer ganzen Karosse sind verschiedene Arbeitsschritte erforderlich. Einen Überblick bietet Darstellung 6.

1 Vgl. Volkswagen AG, Intranet : VW-Rundgang, o. J.



Quelle: Eigene Darstellung unter Verwendung einer Messgerät-Abbildung der Brown & Sharpe Inc. Website

Darstellung 6: Schritte einer Messung

Ausgangspunkt für eine Prüfung sind die Daten des Computermodells des entsprechenden Teiles (CAD-Daten). Unter Berücksichtigung von festgelegten Messprinzipien und Prüfplänen werden Messprogramme (DMIS-Input-Format) erstellt. Dabei wird zum Beispiel das Plugin IQ-MESS eingesetzt, das die CAD-Software CATIA[1] erweitert und den Programmierer unterstützt, indem sie die Auswahl von Merkmalen im CAD-Modell erlaubt und bestimmte DMI-Quellcodeteile generiert, wobei unter anderem die Koordinaten des ausgewählten Merkmales einfließen. Die Messprogramme werden an das KMG übertragen und von diesem ausgeführt. Während der Ausführung protokolliert das KMG unter anderem die Messergebnisse in einer Datei im DMIS-Output-Format. Die Datei wird anschließend rechnergestützt analysiert. Ihre Daten werden in einer Datenbank abgelegt und zur Auswertung abgefragt und aufbereitet (Produkt QUIRL). Fertige Analyse-Berichte nimmt die Software QS-INFO auf, die sie im Intranet verfügbar macht und die Zugriffsberechtigungen verwaltet.

1 Hersteller von CATIA ist die französische Firma Dassault Systems S. A..

1.2.2 Sprachumfang

Das DMIS-Ouput-Format bildet eine Untermenge des DMIS-Input-Formates. Die hier zulässigen Anweisungen sind eigentlich nicht als Anweisungen zu verstehen, sondern als Protokoll der Programmausführung, denn bestimmte Anweisungen des Messprogramms werden in die DMO-Datei übernommen. Eine DMO-Datei soll jedoch nicht ausgeführt sondern ausgewertet werden. Aufgrund seiner Protokollfunktion hat das Output-Format teilweise einen vom Input-Format abweichenden Aufbau.

In Messprogrammen können Variablen verwendet werden. Da DMIS typsicher ist, müssen diese vor der ersten Verwendung deklariert werden. Eingebaut sind logische, numerische sowie Zeichen- und Vektor-Typen.[1] Darüber hinaus können mit bestimmten Anweisungen komplexere Datengebilde wie zum Beispiel geometrische Figuren im Speicher abgelegt werden. Diesen wird ein Labelname zugeordnet, über den das Gebilde referenziert werden kann.[2] Typpräfixe sollen auch bei Labelnamen Typsicherheit gewährleisten.[3] Überall im Programm kann an Stelle eines Wertes auch eine Variable oder ein Ausdruck stehen. Ausdrücke können beliebig tief geschachtelt werden.[4]

Zur Ablaufsteuerung stehen die üblichen Konstrukte wie bedingte Sprünge (`IF`), Zählschleifen (`DO`, Java: `for`) und Mehrfachauswahlen (`SELECT`, Java: `switch`) zur Verfügung. Darüber hinaus sind auch unbedingte Sprünge möglich (`JUMPTO`).[5] Ferner können Makros definiert werden. Diese dürfen eine Parameterliste besitzen und können auch in anderen Dateien definiert sein.[6]

1 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 136 f.

2 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 16

3 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 14

4 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 15

5 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 32 f.

6 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 33

2 Compilerbau

Um eine Aussage über die Konformität einer Datei zum DMI-Standard treffen zu können, wird der DMIS-Checker einen Text parsen und prüfen. Dies sind elementare Techniken des Compilerbaus. Daher sollen in diesem Abschnitt einige Grundlagen des Compilerbaus dargestellt werden, die für das Verständnis des Kapitels zur Projektdurchführung notwendig sind.

2.1 Sprach-Definition

Eine *Sprache* ist eine Menge bestimmter Zeichenfolgen. Sie kann durch Angabe aller erlaubten Zeichenfolgen oder geeigneter Regeln beschrieben werden. Formale Sprachen definieren Regeln, mit denen alle zulässigen Zeichenfolgen einer Sprache gebildet werden können. In diesem Zusammenhang wird eine beliebige Zeichenfolge als *Wort* bezeichnet. Handelt es sich um eine in einer Sprache zulässige Zeichenfolge, stellt sie einen *Satz* der Sprache dar.[1]

Nach *Louden* stehen die Grammatiken der *Chomsky*-Hierarchie für verschiedene Grade der Berechenbarkeit. Sie umfasst die folgenden vier Typen von Grammatiken:

Reguläre Grammatiken sind am wenigsten mächtig, das heißt, sie können nur relativ einfache Gebilde, wie zum Beispiel Zahlen oder Bezeichner, beschreiben. Die entsprechenden Automaten zur Erkennung der Gebilde sind jedoch am effizientesten.

Kontextfreie Grammatiken sind in der Lage, die Syntax einer Sprache zu beschreiben. Sie legen den Aufbau eines Nonterminals fest, unabhängig davon, wo es verwendet wird. Über die Möglichkeiten von regulären Grammatiken hinaus ist Rekursion zulässig. Die gesteigerte Komplexität erhöht jedoch auch den Aufwand der Erkennung.

Kontextsensitive Grammatiken ermöglichen die Beschreibung semantischer Regeln einer Sprache, indem bei einer Regel angegeben werden kann, in welchem Kontext sie angewendet werden darf, das heißt, welches Terminal zum Beispiel vor dem Nonterminal stehen muss. Dadurch kann beispielsweise festgelegt werden, dass eine Variable vor ihrer Benutzung deklariert werden

1 Vgl. Baeumle, P./Alenfelder, H., Compilerbau, 1995, S. 15

muss. Kontextsensitive Grammatiken sind zwar mächtiger als kontextfreie, die Erstellung eines entsprechenden Parsers ist jedoch viel schwieriger, weil die Abhängigkeit von einem Kontext die Komplexität erhöht.

Unbeschränkte Grammatiken unterliegen keinen Einschränkungen, werden aber auch bislang nicht im Compilerbau eingesetzt.[1]

Die lexikalische und syntaktische Struktur einer Sprache werden üblicherweise formal mittels regulärer Ausdrücke beziehungsweise kontextfreier Grammatiken festgelegt. Die Semantik wird hingegen selten formal, sondern meist umgangssprachlich beschrieben.[2]

2.2 Compiler

Laut *Aho*, *Sethi* und *Ullmann* ist ein Compiler ein Programm, das ein in einer bestimmten Quellsprache geschriebenes Programm liest und es in ein äquivalentes Programm einer anderen Sprache, der Zielsprache, übersetzt. Dabei besteht eine wichtige Teilaufgabe des Compilers darin, Fehler, die im Quellprogramm enthalten sind, zu melden.

Seit den 1950er Jahren hat sich das Wissen über Aufbau und Entwicklung von Compilern stark weiterentwickelt. Es wurden systematische Techniken für viele der wichtigen Aufgaben bei der Compilierung entdeckt und Software-Werkzeuge entwickelt. Auch die Weiterentwicklung bei den Programmiersprachen hat ihren Teil zur heutigen Compilererstellung beigetragen. So betrug der Aufwand für die Entwicklung des ersten Fortran-Compilers 18 Mannjahre, während heute „ein funktionstüchtiger Compiler selbst als Studentenarbeit in einem einsemestrigen Compilerbau-Praktikum implementiert werden“[3] kann.[4]

Es gibt zahlreiche Compiler für verschiedenste Quell- und Zielsprachen, die jedoch alle einen ähnlichen Aufbau besitzen. Ein Compiler setzt sich aus zwei Teilen zusammen. Der erste Teil übernimmt die Analyse des Quellprogramms,

1 Vgl. Louden, K. C., Compiler Construction, 1997, S. 131 - 133

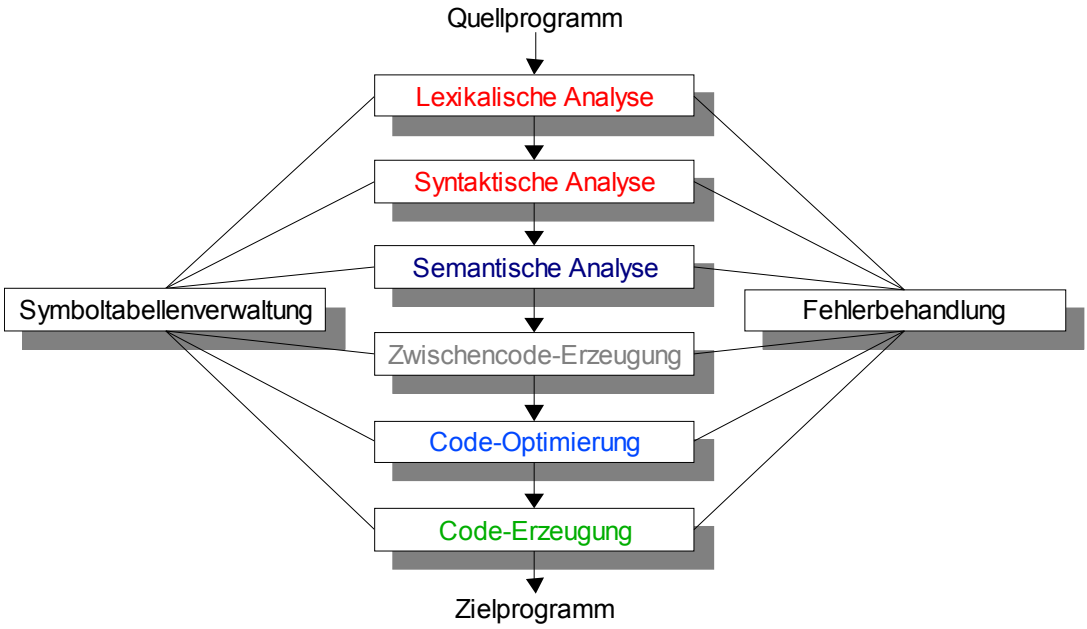
2 Vgl. Louden, K. C., Compiler Construction, 1997, S. 16

3 Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 2

4 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 1 f.

der zweite die Synthese des Zielprogramms. Diese Teile können in Phasen unterteilt werden.

2.2.1 Phasen eines Compilers



Quelle: In Anlehnung an Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 12 und Baeumle, P./Alenfelder, H., Compilerbau, 1995, S. 5

Darstellung 7: Phasen eines Compilers

Eine typische Aufteilung eines Compilers in Phasen zeigt Darstellung 7. Jede Phase überführt das Quellprogramm von einer Darstellung in eine andere. In der Praxis können einige Phasen zusammengefasst werden, wobei die entsprechenden Zwischendarstellungen eventuell nicht mehr erzeugt werden. Die Symboltabellenverwaltung und die Fehlerbehandlung können nicht einer bestimmten Phase zugeordnet werden, sondern sie müssen vielmehr allen Phasen zur Verfügung stehen, so dass sie nicht als Phasen bezeichnet werden können.

Nach *Wilhelm* und *Maurer* befindet sich zwischen lexikalischer und syntaktischer Analyse eine Phase, in der die vom Scanner gelieferten Symbole gefiltert werden. Hier werden zum Beispiel Leerzeichen und Zeilenumbrüche,

die die einzelnen Symbole voneinander getrennt haben, sowie Kommentare und Befehle für den Compiler aus dem Symbol-Strom entfernt.[1]

2.2.1.1 Symboltabellenverwaltung

Die Symboltabelle nimmt die von der lexikalischen Analyse im Quellprogramm gefundenen Bezeichner auf. Außerdem speichert sie zu jedem dieser Bezeichner zusätzliche Informationen wie zum Beispiel Datentyp und Gültigkeitsbereich. Diese können jedoch meist bei der lexikalischen Analyse dem richtigen Bezeichner noch nicht zugeordnet werden. Daher wird die Symboltabelle nach und nach während der Analysephasen gefüllt. Je nach Bedarf lesen die Phasen Informationen aus. Wegen der zahlreichen Zugriffe muss insbesondere ein schnelles Auffinden eines bestimmten Bezeichners in der Tabelle möglich sein.[2]

2.2.1.2 Fehlerbehandlung

Nach *Aho*, *Sethi* und *Ullmann* können Fehler in jeder Phase auftreten und müssen dem Benutzer gemeldet werden. Da die Phasen aufeinander aufbauen, wirkt sich ein aufgetretener Fehler auf die folgenden Phasen und letztendlich auf das Zielprogramm aus, so dass es nahe liegt, beim Auftreten eines Fehlers die gesamte Compilierung abzubrechen. Hilfreicher für den Benutzer ist jedoch ein Compiler, der versucht, hinter der Stelle, an der der Fehler aufgetreten ist, wieder aufzusetzen.[3] Dadurch hat der Benutzer zum Beispiel die Möglichkeit, sich wiederholende Fehlerursachen zu erkennen. Es können aber auch Folgefehler gefunden werden, die ein Benutzer als solche erkennen muss, damit er nicht nach deren Ursache sucht. Wurde zum Beispiel eine Variable falsch deklariert, würde sie nicht in die Symboltabelle eingetragen und wäre infolgedessen unbekannt, wenn sie an anderer Stelle benutzt wird. Im schlimmsten Fall findet der Compiler keinen Punkt zum wieder Aufsetzen. Dies wäre beispielsweise der Fall, wenn im Quellcode einer Java-Klasse die öffnende geschweifte Klammer fehlen würde.

---

1 Vgl. Wilhelm, R./ Maurer, D., Übersetzerbau, 1997, S. 226 f.

2 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 13

3 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 14

„Die Phasen der syntaktischen und semantischen Analyse behandeln gewöhnlich einen Großteil der Fehler, die ein Compiler überhaupt entdecken kann. Die lexikalische Phase kann solche Fehler erkennen, bei denen die in der Eingabe befindlichen Zeichen kein Symbol der Sprache bilden. Fehler, bei denen der Symbolstrom Strukturregeln (Syntax) der Sprache verletzt, werden von der Syntaxanalyse-Phase entdeckt. Während der semantischen Analyse versucht der Compiler Konstrukte zu erkennen, die zwar syntaktisch korrekt strukturiert sind, die aber für die darin enthaltenen Operationen keinen Sinn ergeben. Das ist z.B. der Fall, wenn man versucht, zwei Bezeichner zu addieren, von denen der eine ein Array-Name ist und der andere der Name einer Prozedur.“[1]

2.2.1.3 Lexikalische Analyse

Die lexikalische Analyse wird auch lineare Analyse oder *scanning* genannt. Dabei wird das Quellprogramm zeichenweise gelesen und der Zeichen-Strom in Symbole oder *tokens* aufgeteilt, das heißt, es wird eine Gruppe von Zeichen zu einem Symbol zusammengefasst. Dies kann zum Beispiel eine Zahl sein, die aus einzelnen Ziffern besteht, oder ein Bezeichner, der aus einzelnen Buchstaben und Ziffern zusammengesetzt ist. Die Zeichen, die die einzelnen Symbole trennen, zum Beispiel Leerzeichen, werden meist entfernt.[2] Der Teil eines Compilers, der die lexikalische Analyse durchführt, wird Scanner genannt.

Der Text eines Symbols heißt *Lexem*. Die Lexeme von Bezeichnern, zum Beispiel Variablennamen, werden in eine Symboltabelle eingetragen.[3]

2.2.1.4 Syntaktische Analyse

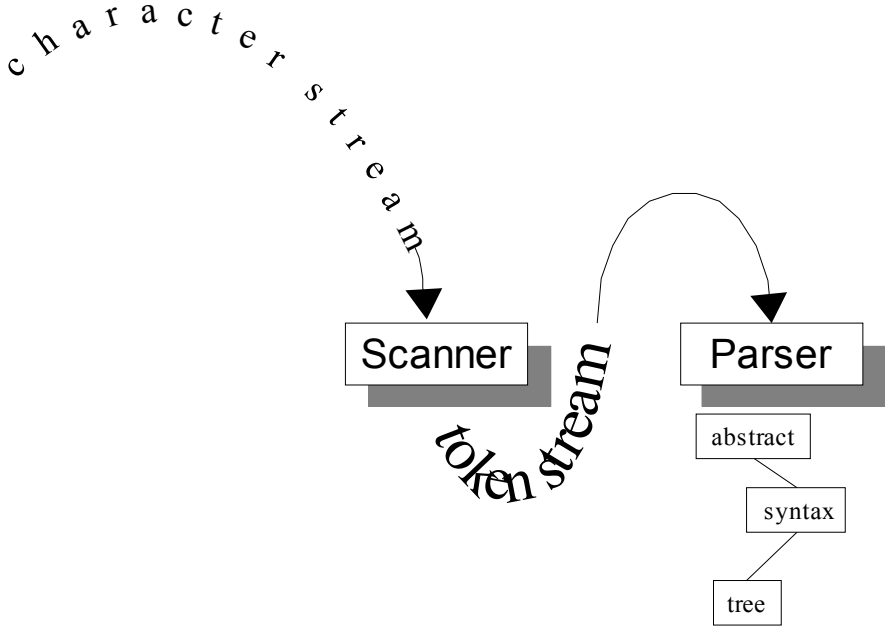
Die Phase der syntaktischen Analyse wird auch als hierarchische Analyse oder *parsing* bezeichnet. Sie fasst die vom Scanner gelieferten Symbole zu grammatikalischen Sätzen zusammen. Diese werden im allgemeinen durch eine Baumdarstellung dargestellt, die als Parse-Baum (parse tree) bezeichnet

1 Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 14

2 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 6

3 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 14

wird.[1] Alternativ ist der Begriff *Syntaxbaum* zu finden. Nach *Louden* ist ein Syntaxbaum ein abstrahierter Parse-Baum, weswegen er auch als abstrakter Syntaxbaum (abstract syntax tree) bezeichnet wird. Demnach enthält der Parse-Baum detaillierte Informationen in Form von Knoten über das Parsen an sich. Die Abstraktion besteht darin, dass bestimmte Knoten, die keine konkreten Konstrukte der Quellsprache darstellen und nur einen Nachfolger haben, entfernt werden.[2] Die syntaktische Analyse wird vom Parser durchgeführt.



Quelle: Eigene Darstellung

Darstellung 8: Weg eines Textes durch lexikalische und syntaktische Analyse

Darstellung 8 veranschaulicht das Zusammenspiel von Scanner und Parser. Der Scanner liest einen Zeichen-Strom und erzeugt einen Symbol-Strom, den der Parser bei der syntaktischen Analyse in eine Baumdarstellung umwandelt.

„Das gerade zu untersuchende Symbol wird oft als Lookahead-Symbol bezeichnet.“[3] Anhand dieses Symbols wird bei Alternativen entschieden, welche gewählt wird. Es ist aber auch ein Lookahead von mehreren Symbolen

1 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 7

2 Vgl. Louden, K. C., Compiler Construction, 1997, S. 9

3 Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 52

möglich. Beginnen beispielsweise zwei Alternativen einer Regel bei einem Lookahead von k=1 mit dem gleichen Symbol, kann anhand dieses einen zur Verfügung stehenden Symbols nicht entschieden werden, welche der beiden Möglichkeiten die richtige ist. Es wäre keine deterministische Entscheidung möglich. Abhilfe kann hier das Erhöhen des Lookaheads oder eine Umformulierung der Regel schaffen. Beides muss jedoch nicht unbedingt zum Ziel führen. In dem Fall handelt es sich nicht um eine kontextfreie Grammatik, so dass eine Verallgemeinerung vorgenommen werden muss, bei der die semantischen Informationen verloren gehen, die in der Regel enthalten sind.

2.2.1.5 Semantische Analyse

Die semantische Analyse untersucht das Quellprogramm auf semantische Fehler. Dazu nutzt sie die Baumdarstellung, die der Parser erzeugt hat. Eine wesentliche Aufgabe sind Typüberprüfungen.[1]

Die Semantik eines Programms ist seine Bedeutung. Sie legt unter anderem das Verhalten zur Laufzeit fest. Verstöße gegen diesen Teil der Semantik können von einem Compiler nicht festgestellt werden, weil er das Programm nicht ausführt. Die restlichen Regeln können jedoch überprüft werden. Dieser Teil wird als statische Semantik bezeichnet. Darunter fallen Typprüfungen.[2]

2.2.1.6 Zwischencode-Erzeugung

Manche Compiler überführen den Syntaxbaum in eine Zwischendarstellung. Diese ist ein Schritt in Richtung Zielprogramm, und „sollte leicht zu erzeugen und leicht ins Zielprogramm zu übersetzen sein“[3]. Bei ihrer Erstellung werden einige Gegebenheiten der Zielsprache berücksichtigt, wie zum Beispiel die Präzedenz von Operatoren, so dass die Komplexität bei der Erzeugung des Zielprogramms verringert wird.[4]

Holmes nennt als Grund für die Erzeugung von Zwischencode die Entkopplung von Analyse- und Synthese-Teil. Der Zwischencode dient also als Schnittstelle. Dadurch bleibt optimaler Weise der nicht geänderte Teil von

1 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 9

2 Vgl. Louden, K. C., Compiler Construction, 1997, S. 9 f.

3 Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 17

4 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 17

Änderungen im anderen Teil unberührt.[1] Die Analyse-Phase „ist (im Idealfall) unabhängig von Eigenschaften der Zielsprache und Zielmaschine“[2].

2.2.1.7 Code-Optimierung

Die Aufgabe dieser Phase ist die Verbesserung des Zwischencodes. Dazu wird dieser untersucht und nach bestimmten Regeln modifiziert. Zum Beispiel werden einzelne Befehle zusammengefasst, so dass letztendlich effizienterer Zielcode erstellt werden kann. Auf diese Phase entfällt bei optimierenden Compilern ein beträchtlicher Teil der Übersetzungszeit.[3]

Laut *Holmes* hat die Code-Optimierung einen großen Einfluss auf die Qualität des Zielcodes. Er betont, dass eine Optimierung nicht zwangsläufig zum Optimum führt, und gibt als Ziele der Optimierung neben einem effizienten Laufzeitverhalten des Zielcodes auch einen geringen Umfang an. Ferner beschreibt er Optimierungen, bei denen kleine Segmente des Zielcodes nach Mustern ineffizienten Codes durchsucht werden.[4]

2.2.1.8 Code-Erzeugung

Die letzte Phase erzeugt die eigentliche Ausgabe des Compilers. Hierbei wird Speicherplatz für alle verwendeten Variablen zugeordnet und jede Instruktion der Zwischendarstellung in eine funktionserhaltende Folge von Befehlen der Zielsprache übersetzt.[5]

2.2.2 Methoden zur Syntaxanalyse

„Die meisten Methoden zur Syntaxanalyse lassen sich in zwei Klassen einteilen, in die sogenannten [!] *Top-Down*- und *Bottom-Up*-Methoden. Die Namensgebung spiegelt die Richtung wider, in der die Knoten des Parse-Baums konstruiert werden. Bei der erstgenannten Methode beginnt man die Konstruktion mit der Wurzel und führt sie in Richtung der Blätter fort; bei der zweiten Methode beginnt man die Konstruktion an den Blättern und arbeitet sich zur Wurzel hoch. Die Popularität von Top-Down-Parsern hat ihren Grund

1 Vgl. Holmes, J., Object-Oriented Compiler Construction, 1995, S. 6

2 Wilhelm, R./ Maurer, D., Übersetzerbau, 1997, S. 225

3 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 18

4 Vgl. Holmes, J., Object-Oriented Compiler Construction, 1995, S. 6

5 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 18

darin, daß [!] es mit Top-Down-Methoden leichter ist, effiziente Parser von Hand zu erstellen. Auf der anderen Seite läßt [!] sich mit Bottom-Up-Methoden eine größere Klasse von Grammatiken und Übersetzungsschemata behandeln. Aus diesem Grund benutzen Software-Werkzeuge zur automatischen Parser-Generierung überwiegend Bottom-Up-Methoden."[1]

2.2.3 Parser-Generatoren

Zu den Werkzeugen, die die Implementierung eines Compilers unterstützen, und damit zur Vereinfachung der Entwicklung eines Compilers beitragen, gehören die Parser-Generatoren. Diese generieren meist auch einen passenden Scanner, da ein Parser auf den Token-Strom des Scanners angewiesen ist, und entsprechend eng mit diesem zusammenarbeitet.

Die Sprache, für die ein Parser erzeugt werden soll, wird in der Regel mit Hilfe einer kontextfreien Grammatik beschrieben.[2] Der Scanner könnte ebenfalls aus einer solchen Grammatik generiert werden, aber die meisten Generatoren nutzen hierfür reguläre Ausdrücke.

1 Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 50 f.

2 Vgl. Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 27

C Projekt-Durchführung

Dieses Kapitel stellt den Schwerpunkt dieser Arbeit dar. Es beschreibt alle Phasen des Projektes.

Zu Beginn werden in der Analysephase die fachlichen Anforderungen herausgearbeitet. Darauf aufbauend wird im Abschnitt Entwurf die Architektur entwickelt. Die Umsetzung in Quelltext beschreibt der Abschnitt über die Implementierung. Abschließend setzt sich die Testphase mit möglichen Fehlerquellen auseinander.

1 Analyse

Dieses Kapitel stellt zunächst die aktuelle Situation dar und beschreibt dann die Rahmenbedingungen, um daraus die fachlichen Anforderungen abzuleiten. Das Kapitel endet mit einer Betrachtung der Vorteile, die der VOLKSWAGEN AG durch das Projekt entstehen.

1.1 Heutige Situation

Aktuell ist die DMIS-Version 04.0. In der Beschreibung des Standards ist eine Grammatik – in EBNF notiert – enthalten. Diese beruht auf einer DMIS 03.0-Grammatik, die ursprünglich im Hause Volkswagen erstellt wurde. DMIS Version 05.0 ist in der Entwicklung.

Für die vorherige Version 03.0 von DMIS existiert bereits ein Prüfwerkzeug. Dies wurde in der Programmiersprache C entwickelt. Zum Lesen der zu prüfenden DMIS-Datei verwendet es einen Parser, der mit Hilfe der UNIX-Tools *lex* und *yacc* erzeugt wurde. Die Änderungen und Erweiterungen von DMIS 03.0 zu DMIS 04.0 einzupflegen, gestaltet sich jedoch schwierig, da die Grammatik, die zur Erzeugung des Parsers für das Prüfwerkzeug benutzt wurde, in der spezifischen Notation des Parser-Generators vorliegt. Es wäre also ein händischer Abgleich mit der EBNF-Grammatik der neuen Version erforderlich, der bei jeder kommenden Version erneut durchzuführen wäre. Dazu sind umfangreiche Kenntnisse im Compilerbau nötig.

Für die Datenaufbereitung durch das Produkt QUIRL wird ein Parser benutzt, der auf dem Parser des Prüfwerkzeuges basiert. Der in die Grammatik eingebettete C-Quelltext wurde jedoch wesentlich verändert, um die DMIS-Daten in ein Zwischenformat zu konvertieren. Deswegen wäre auch hier wiederum ein manueller Abgleich nötig, um Dateien neuer DMIS-Versionen in QUIRL importieren zu können.

Des Weiteren gibt es eine Software, die „DMI-DMO-Abgleich“ genannt wird und einen Abgleich zwischen einer DMI-Datei und einer korrespondierenden DMO-Datei ermöglicht. Sie prüft, ob für alle Befehle, die in der DMI-Datei enthalten sind, entsprechende DMO-Statements vom KMG ausgegeben worden sind. Wichtig für diesen Test ist jedoch, dass die Dateien

für sich jeweils zumindest syntaktisch fehlerfrei sind. Da kein Prüfwerkzeug für die aktuelle DMIS-Version vorliegt, wurde diese Software nicht weiterentwickelt. Sie wird jedoch zur Beurteilung der Qualität von KMGs ebenso benötigt wie ein DMIS-Prüfwerkzeug, das mit relativ geringem Aufwand an aktuelle Versionen angepasst werden kann.

Da es Werkzeuge gibt, die XML-Dateien auf Korrektheit prüfen, stellt sich die Frage, warum die Prüfprogramme und -protokolle nicht im XML-Format gespeichert und die vorhandenen Prüfwerkzeuge benutzt werden. Dieser Weg führt jedoch zur Zeit nicht zum Ziel, da die vorhandenen KMGs XML nicht unterstützen. Es wäre also eine Übersetzung von DMIS nach XML nötig, die einen Parser für DMIS ebenso erfordern würde, wie der angestrebte DMIS-Checker. Fraglich ist auch, ob die Semantik von DMIS in einer DTD oder einem XML-Schema formuliert werden kann. Dann müsste ein Übersetzungs-progamm jedoch die Semantik einer DMIS-Datei erkennen und in entsprechende XML-Konstrukte umsetzen. Das Übersetzungsprogramm würde im Prinzip bereits während der Übersetzung sämtliche Prüfungen durchführen, so dass der DMIS-Checker offenbar die Mindest-Funktionalität eines solchen Übersetzungsprogramms darstellt.

1.2 Vorgaben

In der Abteilung „Informationstechnik Elektronik- und Qualitätsinformation – Messen und Prüfen“ und bei den Kunden (Abteilungen, die die Software einsetzen) werden verschiedene Hardware- und Software-Plattformen eingesetzt. CAD-Anwendungen werden beispielsweise unter verschiedenen UNIX-Systemen betrieben, während die Java-Programmierung unter Windows durchgeführt wird. Daher soll auf Plattformunabhängigkeit geachtet werden.

Als Programmiersprache wird überwiegend Java genutzt, weil damit gute Erfahrungen gemacht worden sind. Das Back-End von QUIRL und das Werkzeug zum DMI-DMO-Abgleich sind dementsprechend in Java programmiert. Da der DMIS-Checker eng mit diesen Programmen zusammenarbeiten wird, soll er ebenfalls in Java implementiert werden. Ein ebenso wichtiger Grund für den Einsatz von Java ist, dass das Know-How zum Pflegen einer Java-Anwendung in der Abteilung vorhanden ist.

Hingegen ist zur Zeit nur wenig Wissen über den Compilerbau in der Abteilung vorhanden. Daher sollen die Anpassungen für kommende DMIS-Versionen möglichst wenig solchen Wissens erfordern. Dieses Ziel unterstützen Parser-Generatoren, weil ihre Anwendung keine detaillierten Kenntnisse über den Aufbau eines Parsers voraussetzt. Es wird ein Modell – in Form einer Grammatik – anstatt des Quelltextes gepflegt, das eine fachliche Abstraktion darstellt. Anhand des Modells kann der Parser zentral an Änderungen des Dateiformates angepasst werden. Ein weiterer Grund für den Einsatz eines Generators ist der Umfang der DMIS-Grammatiken, denn es gibt für die beiden DMIS-Formate jeweils eine DMIS-Grammatik. Die Grammatik für das Eingabe-Format (DMI) füllt mit ihren 460 Regeln ungefähr 55 DIN-A4-Seiten, die für das Ausgabe-Format (DMO) mit 255 Regeln beinahe 40. Die Grammatiken sind so umfangreich, weil sie nicht nur die Syntax beschreiben, sondern auch semantische Informationen enthalten. So werden zum Beispiel nicht nur Zahlen und Bezeichner unterschieden, sondern sogar Ganz- und Dezimalzahlen. Es ist also aus der Grammatik ersichtlich, welcher Datentyp wo zulässig ist. Aber die Zahlen werden noch weiter unterteilt, nämlich in Wertebereiche wie zum Beispiel kleiner Null, gleich Null oder größer Null. Ferner sind hier im Gegensatz zu einer Grammatik einer universell einsetzbaren Sprache wie C oder Java alle Anweisungen mit allen möglichen Parameter-Kombinationen angegeben. Durch diese über die reine Syntax hinaus gehenden Angaben steigt die Komplexität der Grammatik wesentlich. Ob all diese Informationen in einen Parser übernommen werden können, wird sich bei der Implementierung zeigen. Je mehr jedoch aus der Grammatik automatisch in einen Parser übernommen und damit in DMIS-Dateien geprüft werden kann, desto wertvoller werden die Ergebnisse des DMIS-Checkers.

1.3 Anforderungen

Es gibt keinen Parser-Generator, der nur aus einer reinen EBNF-Grammatik einen Parser erzeugen kann. Es werden immer zusätzliche Informationen benötigt. Ferner weicht die Notation der Grammatik bei jedem Werkzeug mehr oder weniger vom EBNF-Standard ab. Aber auch die EBNF-Grammatik aus

der DMIS-Standard-Dokumentation entspricht nicht dem EBNF-Standard. Daher muss die Ausgangsgrammatik modifiziert werden. Entsprechend besteht die Aufgabenstellung aus zwei Teilen: dem Prüfwerkzeug für DMIS-Dateien selbst und einem Grammatik-Generator, der aus der EBNF-Grammatik des Standards eine Grammatik für den verwendeten Parser-Generator im entsprechenden Format erzeugt. Ob ein solcher Grammatik-Generator machbar ist, muss geprüft werden. Ist dies nicht der Fall, ist eine Grammatik zum Generieren eines Parsers von Hand zu erstellen.

Kenntnisse im Compilerbau sollen zur Anwendung und Pflege des Prüfwerkzeuges möglichst nicht erforderlich sein. Entsprechende Arbeiten sollte der Grammatik-Generator einem Benutzer also abnehmen. Einige dieser Aufgaben sind jedoch sehr komplex, so dass ein vollkommen selbständiger Generator sicherlich über den Umfang dieser Arbeit hinaus ginge. Ziel ist daher ein Werkzeug zu entwickeln, das dem Benutzer hilft, eine Grammatik zu erstellen, indem es Funktionen anbietet und Hinweise auf mögliche Probleme gibt. Dieses könnte nach und nach erweitert werden. Zum minimalen Funktionsumfang gehört das Übersetzen aus der EBNF-Notation in die spezifische Notation des Parser-Generator. Um die Ausgangs-Grammatik überhaupt auf irgendeine Weise verarbeiten zu können, muss sie geparst werden. Ist das Parsen abgeschlossen, stellt die Erzeugung einer Grammatik in einem anderen Format kein großes Problem dar. Dieses Tool besteht also hauptsächlich aus einem einfachen Compiler, denn die Übersetzung stellt ja gerade den eigentlichen Zweck von Compilern dar. Wäre das Parsen nicht möglich, wäre der gesamte Grammatik-Generator hinfällig.

Neben der Übersetzung sollten einige weitere Funktionen angeboten werden, die dem Benutzer unangenehme Fleißarbeiten bei der Anpassung der EBNF-Grammatik an die Erfordernisse des Parser-Generators abnehmen. Dies könnte zum Beispiel die Suche nach unzulässigen Bezeichnern und deren Ersetzung sein, damit der Parser-Quelltext compiliert werden kann. Können mögliche Probleme nur lokalisiert, nicht aber behoben werden, sollte der Benutzer darauf hingewiesen werden.

Darüber hinaus wäre die Generierung einer neuen EBNF-Grammatik nach der Bearbeitung von Vorteil, um Verbesserungsvorschläge zur Formulierung der Grammatik machen zu können. Auch die Aufbereitung der Grammatik in einem leichter lesbaren Format bietet sich an.

Folgendes soll der Grammatik-Generator leisten:

- **Übersetzung** von EBNF ins Parser-Generator-spezifische Format
- **Durchführung von Änderungen** an der Grammatik
- **Unterstützung** des Benutzers beim Auffinden von möglichen Problemen
- **Generierung von Dokumentation**

Darüber hinaus soll er zukünftig leicht erweiterbar sein.

Der DMIS-Checker muss auch für kommende DMIS-Versionen angepasst werden können. Dazu soll nur aus einer entsprechenden Grammatik ein Parser generiert werden, nicht aber der DMIS-Checker selbst angefasst werden müssen. Eine Anpassung oder Erweiterung darf ebenfalls nur möglichst wenig Kenntnisse im Compilerbau erfordern.

Ferner müssen alle lexikalischen und syntaktischen Fehler festgestellt werden, soweit diese aus der Grammatik hervorgehen. Zusätzliche semantische Prüfungen werden nur beispielhaft implementiert werden. An Hand dieser sollen später weitere umgesetzt werden können, so dass auf gute Erweiterbarkeit geachtet werden muss. Bei einer Prüfung müssen so viele Fehler wie möglich gefunden werden, damit eine vernünftige Aussage über die Korrektheit der gesamten Datei gemacht werden kann. Würde beim ersten Fehler abgebrochen, wäre das Programm praktisch wertlos, denn um einen Gesamteindruck der Datei zu erhalten, müsste jeder gefundene Fehler korrigiert und eine erneute Prüfung angestoßen werden. Ähnlich wichtig sind aussagekräftige Fehlermeldungen für die Nützlichkeit des Programms. Kann ein Anwender das Ergebnis nicht nachvollziehen, ist die Software ebenfalls nutzlos.

Der generierte Parser soll ohne Änderungen auch von anderen Programmen genutzt werden können, zum Beispiel zum Importieren von Daten nach QUIRL. Und auch der DMIS-Checker selbst muss leicht durch andere

Programme aufrufbar sein, denn er wird vom DMI-DMO-Abgleich verwendet werden. Um DMIS-04.0-Daten nach QUIRL importieren zu können, während gleichzeitig ein KMG auf DMIS-05.0-Fähigkeit getestet wird, müssen Parser für verschiedene DMIS-Versionen parallel zur Verfügung stehen können.

Zusammenfassend kann gesagt werden, dass das primäre Ziel des Projektes, gleich auf welchem Weg es erreicht wird, das Prüfwerkzeug „DMIS-Checker“ ist. Es soll folgende Eigenschaften aufweisen:

- **Anpassbarkeit** über eine Grammatik
- **Genauigkeit**, damit alle lexikalischen und syntaktischen Fehler gefunden werden
- **Toleranz**, um nach dem Erkennen eines Fehlers das Prüfen fortsetzen zu können
- **Wiederverwendbarkeit** insbesondere des Parsers
- **Flexibilität**, damit verschiedene DMIS-Versionen ohne erneute Compilierung geprüft werden können
- **Nützlichkeit** durch aussagekräftige Fehlermeldungen
- **Integrierbarkeit** in andere Programme
- **Erweiterbarkeit**, damit semantische Prüfungen ergänzt werden können

1.4 Nutzen für die Volkswagen AG

KMGs sind Hightech-Geräte mit entsprechend hohem Wert. Daher sollte bereits bei ihrer Anschaffung im Rahmen eines DMIS-Fähigkeitstests überprüft werden können, ob und in welchem Maße sie den DMI-Standard einhalten. Andernfalls haben Versprechen des Herstellers und vertragliche Festlegungen keinen Wert. Zu solchen Prüfungen soll der DMIS-Checker herangezogen werden können.

Wegen ihrer hohen Anschaffungskosten werden solche Geräte nicht sehr zahlreich angeschafft, insbesondere da diese nicht direkt zur Wertschöpfung erforderlich sind. Entsprechend sorgsam muss mit der vorhandenen Kapazität umgegangen werden, um einen möglichst großen Nutzen aus dem investierten Kapital ziehen zu können. Die Auslastung kann zum Beispiel mit TERAMESS, einem Produkt der Abteilung, gesteuert und überwacht werden. Zur

Vermeidung von Verschwendung der knappen Messraumzeit kann der DMIS-Checker beitragen, indem DMIS-Messprogramme vor ihrem Einsatz geprüft und Fehler korrigiert werden. *Wocke* bestätigt, dass die „kostenintensive Meßmaschine [!] .. am wirtschaftlichsten eingesetzt [ist], wenn sie – möglichst sogar im Mehrschichtenbetrieb – ständig fertige Meßprogramme [!] abfährt und nicht ... "mißbraucht" [!] wird".[1]

Allgemein führt die Einhaltung von Standards zu verschiedenen Vorteilen für ein Unternehmen, wie aus einer im Auftrag des Deutschen Instituts für Normung durchgeführten Studie hervorgeht. Hierzu gehören Kostenersparnisse durch das Wegfallen von Anpassungen an verschiedene Formate und ein größeres Angebot an Kooperationspartnern und Zulieferern.[2] Dadurch können Abhängigkeiten von einzelnen Firmen verhindert werden.[3]

Ferner ergeben sich einem an der Normungsarbeit beteiligten Unternehmen meist Kosten- und Wettbewerbsvorteile, wenn zum Beispiel nationale zu internationalen Normen werden oder Normen in Gesetze einfließen.[4]

1 Wocke, P. M., Koordinatenmeßmaschinen, 1993, S. 1

2 Vgl. DIN, Gesamtwirtschaftlicher Nutzen der Normung, 2000, S. 14

3 Vgl. DIN, Gesamtwirtschaftlicher Nutzen der Normung, 2000, S 16

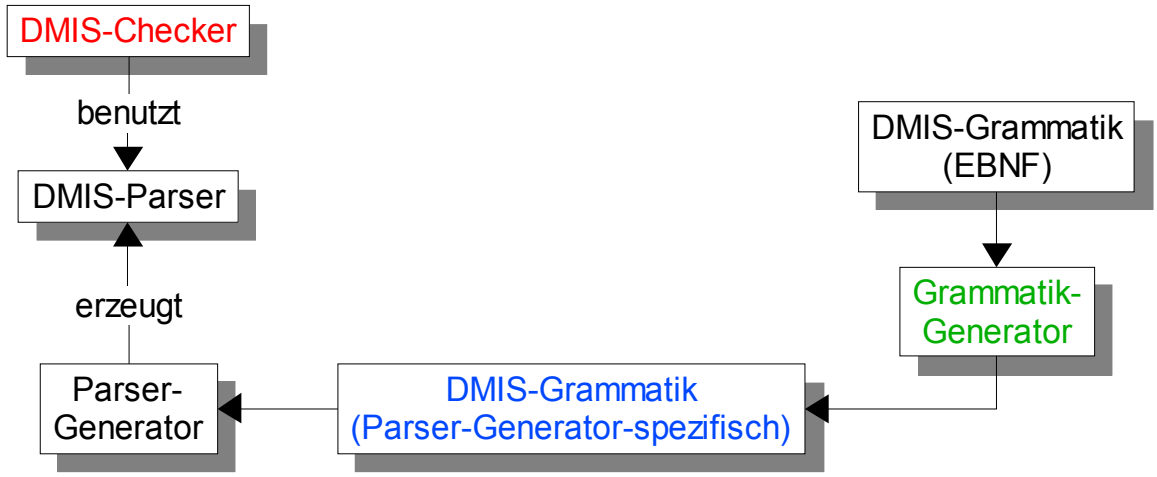
4 Vgl. DIN, Gesamtwirtschaftlicher Nutzen der Normung, 2000, S. 11

2 Entwurf

Das Kapitel beginnt mit einem groben Überblick über das Zusammenspiel des DMIS-Checkers mit den Generatoren. Um die Gegebenheiten des gewählten Parser-Generators und der von diesem generierten Parser im Entwurf berücksichtigen zu können, ist die Auswahl eines geeigneten Parser-Generators dem eigentlichen Entwurf vorangestellt. Anschließend wird der Entwurf sowohl für den DMIS-Checker als auch für den Grammatik-Generator beschrieben.

2.1 Überblick

Darstellung 9 zeigt, wie der DMIS-Checker, ein noch auszuwählender Parser-Generator und der Grammatik-Generator zusammenarbeiten.



Quelle: Eigene Darstellung

Darstellung 9: Gesamtzusammenhang

Die DMIS-Grammatik im Format des Parser-Generators stellt einen wichtigen Zwischenschritt auf dem Weg zum DMIS-Checker dar. Diese Grammatik ist die Schnittstelle zwischen den beiden Projektteilen. Ob sie von einem Werkzeug generiert oder manuell erstellt wurde, ist für die Nutzung im DMIS-Checker unerheblich.

Der aus dieser Grammatik generierte Parser wird den Hauptteil der Prüfung einer DMIS-Datei übernehmen. Daher hängt der Nutzen des DMIS-Checkers direkt mit der Qualität dieser Grammatik und des verwendeten Parser-

Generators zusammen. Damit steigen auch die Anforderungen an den Grammatik-Generator.

2.2 Auswahl eines Parser-Generators

Nach der Festlegung der Kriterien werden die Kandidaten beschrieben. Darauf folgen eine Bewertung und die Auswahl. Der Abschnitt endet mit einer Beschreibung der für den Entwurf wichtigen Eigenheiten des gewählten Parser-Generators.

2.2.1 Kriterien

Aus der Analyse ergibt sich, dass der Parser-Generator die Generierung von Java-Quelltext unterstützen muss. Dies ist unbedingt erforderlich.

Da es eine große Zahl frei verfügbarer Werkzeuge gibt und auch bei diesem Projekt auf die Kosten geachtet wird, sollen kommerzielle Produkte außer Acht gelassen werden.

Wünschenswert ist eine Unterstützung der Baumerzeugung beim Parsen, weil sowohl beim DMIS-Checker als auch beim Grammatik-Generator Bäume benötigt werden. Ohne die Baumdarstellung wäre der DMIS-Parser nicht wiederverwendbar, weil die geparsten Daten sofort weiterverarbeitet werden müssten. Andernfalls gingen sie verloren. Zur sofortigen Verarbeitung sind semantische Aktionen erforderlich, die in die Regeln der Grammatik eingebettet werden müssen. Für jede weitere, andere Verwendung des Parsers müssten also entsprechende semantische Aktionen eingefügt werden. Zwischen diesen Aktionen müsste bei jeder Nutzung des Parsers unterschieden werden. Das passt nicht zum Gedanken der Wiederverwendbarkeit. Semantische Prüfungen, die mehrere DMIS-Anweisungen betreffen, wären ebenfalls nicht realisierbar. Die Übersetzung von EBNF ins Parser-Generator-spezifische Format ist zwar auch ohne Baumdarstellung möglich, aber zusätzliche Funktionen könnten nicht umgesetzt werden. Es könnten semantische Aktionen in die Grammatik hinein geschrieben werden, die einen Baum aus den geparsten Daten erstellen. Dies würde jedoch die Lesbarkeit der Grammatik beeinträchtigen. Außerdem müssten bei jeder Änderung an der Grammatik die Anweisungen zur Baumerzeugung ebenfalls geändert werden.

Damit die Arbeit mit dem Generator nicht unnötig schwierig wird, sollte die Notation der Grammatik gut lesbar sein.

Nicht vergessen werden darf, dass eine lange Nutzungsdauer angestrebt wird. Es sollte sichergestellt sein, dass die Entwicklung und Pflege des Werkzeugs nicht eingestellt wird. Garantieren kann das jedoch niemand. Daher kann nur die aktuelle Situation berücksichtigt werden.

Folgende Kriterien werden zur Entscheidung herangezogen:

- **Generierung von Java-Quelltext**
- **Freie Nutzung**
- **Unterstützung bei der Baumerstellung**
- **Lesbarkeit der Grammatiken**
- **Support**

2.2.2 Vorstellung der Kandidaten

Ein Generator, der direkt die vorliegende Grammatik in EBNF versteht, konnte nicht gefunden werden. Dies ist verständlich, da zum Beispiel bei den betrachteten Generatoren zahlreiche Optionen angegeben werden können oder sogar müssen. Für diese Optionen muss es eine Notation geben. Außerdem stellt die Aufteilung der Regeln auf Scanner und Parser für ein Programm ein schwieriges Problem dar, weil nicht in jedem Fall einfach zwischen lexikalischer und syntaktischer Analyse zu entscheiden ist, wie aus der folgenden Aussage von *Aho*, *Sethi* und *Ullmann* hervorgeht: „Die Trennung zwischen lexikalischer und syntaktischer Analyse ist etwas willkürlich. Im allgemeinen [!] wählen wir die Trennung so, daß [!] die gesamte Analyseaufgabe einfacher wird.“[1] Zur Erzeugung eines Baumes sind zudem semantische Informationen nötig, die in der Grammatik nicht enthalten sind.

Im Folgenden werden die zur Wahl stehenden Werkzeuge vorgestellt. Dabei wird jeweils erläutert, warum das Werkzeug betrachtet wird, ob es in Frage kommt und in welchem Maß die Kriterien erfüllt werden.

1 Aho, A. V./Sethi, R./Ullmann, J. D., Compilerbau, 1999, S. 8

2.2.2.1 lex und yacc

Eine Verwendung von *lex* und *yacc* liegt nahe, da der bereits vorhandene DMIS-Checker diese Werkzeuge nutzt. Aus der Dokumentation ist jedoch ersichtlich, dass die Generierung von Java-Quellcode nicht möglich ist[1]. Daher erübrigen sich weitere Untersuchungen.

2.2.2.2 JavaCC

Der *Java Compiler Compiler* wurde ursprünglich von *Sreeni Viswanadha* und *Sriram Sankar* während ihrer Tätigkeit bei *Sun* entwickelt, wird jedoch mittlerweile auch von der Community gepflegt,[2] da im Juni 2003 daraus ein Open-Source-Projekt wurde.[3] Laut der Website des Projektes ist JavaCC der am weitesten verbreitete Parser-Generator in Verbindung mit Java-Anwendungen.[4]

Die Notation der Grammatiken für JavaCC lehnt sich teilweise an die Syntax von Java an. So steht beispielsweise der Regelkörper in einem Block aus geschweiften Klammern, und der Regelname für Parser-Regeln erinnert mit vorangehendem Rückgabedatentyp und folgenden runden Klammern an eine Methodendeklaration. Da es bei Top-Down-Parsern für jede Regel eine Methode gibt und Attribute hin und her gereicht werden können, erscheint diese Vorgehensweise konsequent. Regeln für den Scanner weichen beim Regelnamen etwas ab. Außerdem müssen hier reguläre Ausdrücke verwendet werden.

Zur Baumerzeugung stellt JavaCC das Werkzeug *JJTree* zur Verfügung. Dies ist ein Präprozessor, der aus einer bezüglich der Baumerzeugung erweiterten Grammatik eine von JavaCC verarbeitbare Grammatik erzeugt.[5]

---

1 Vgl. Lesk, M. E./Schmidt, E., Lex, o. J., Abschnitt „ABSTRACT“

2 Vgl. Norvell, T. S., The JavaCC FAQ, 2003, Abschnitt 1.14

3 Vgl. Norvell, T. S., The JavaCC FAQ, 2003, Abschnitt 1.8

4 Vgl. o. A., javacc: JavaCC Home, o. J., Abschnitt „Description“

5 Vgl. o. A., JavaCC: JJTree Reference Documentation, o. J., Abschnitt „Introduction“

2.2.2.3 CoCo/R

Mit *CoCo/R* habe ich bereits Erfahrungen sammeln können, so dass keine Einarbeitung erforderlich ist. Daher soll auch dieser Generator auf Eignung untersucht werden.

Die erste Version von CoCo/R hat *Hanspeter Mössenböck* im Jahre 1983 entwickelt. Inzwischen gibt es unter anderem Versionen für Java, C, Pascal und C#, die von einzelnen Personen entwickelt wurden.[1] Die Existenz einer C#-Version spricht dafür, dass das Werkzeug zur Zeit noch gepflegt wird, da es sich bei C# um eine relativ neue Programmiersprache handelt. Eine große Community wie zum Beispiel bei JavaCC gibt es offenbar nicht.

Die Grammatiken werden in EBNF notiert,[2] unterscheiden sich jedoch von der ebenfalls in EBNF notierten DMIS-Grammatik aus der Standard-Dokumentation.

Es ist ein Lookahead von nur einem Zeichen, bzw. Token möglich. Außerdem wird die Erzeugung von Bäumen von CoCo/R nicht unterstützt. Der Aufbau eines Baumes müsste also mit semantischen Aktionen gesteuert werden.

2.2.2.4 ANTLR

ANTLR steht für „Another Tool for Language Recognition“. Es ist wie JavaCC weit verbreitet, denn es wird bei vielen akademischen und industriellen Projekten eingesetzt.[3] Initiiert hat das ANTLR-Projekt *Dr. Terrence Parr*, der daran bis heute beteiligt ist. Dabei unterstützen ihn Kollegen und die Community.[4]

Neben Java kann auch C++ oder C# als Sprache des zu generierenden Parsers gewählt werden. Die Unterstützung von Python ist in Vorbereitung.[3] Dass bereits auch relativ neue Sprachen wie C# und Python unterstützt werden, spricht für die Aktivität der Community.

---

1 Vgl. Terry, P. D., Coco/R, 2003, Abschnitt „Introduction“

2 Vgl. Terry, P. D., Coco/R for Java, 2002, Abschnitt „Introduction“

3 Vgl. Parr, T., Why Use ANTLR?, o. J.

4 Vgl. Parr, T., ANTLR Reference Manual, 2003, Abschnitt „What's ANTLR“

Die Grammatiken für ANTLR ähneln der DMIS-Grammatik stark. Die Notation wird in der Dokumentation als EBNF-ähnlich beschrieben.[1] Zusätzliche, ANTLR-spezifische Angaben sind nur dort erforderlich, wo ANTLR von seinem Standard-Verhalten abweichen soll. Der einzige Unterschied in der Notation zwischen Regeln für den Scanner und Regeln für den Parser besteht im Anfangsbuchstaben des Regelnamen. Bei Scanner-Regeln fängt der Name mit einem großen Buchstaben an, bei Parser-Regeln mit einem kleinen Buchstaben. Das heißt, dass die Regeln für den Scanner mit kontextfreien Grammatiken beschrieben werden können und nicht als reguläre Ausdrücke angegeben werden müssen. Daher sind nicht von vornherein manuelle Änderungen an der Grammatik nötig, obwohl diese auch Regeln enthält, die den Aufbau von Symbolen beschreiben.

Die für die Baumerzeugung erforderlichen Angaben werden in die Grammatik geschrieben. Dies beschränkt sich darauf, dass markiert wird, was zur Wurzel eines Unterbaumes werden soll. Aus der Reihenfolge innerhalb der Regel ergibt sich dann, was in den Unterbaum aufgenommen wird.[2]

2.2.3 Bewertung und Auswahl

Die fehlende Unterstützung bei der Baumerzeugung kann CoCo/R nicht durch andere Eigenschaften ausgleichen. Vielmehr bestärkt die Entwicklung durch Einzelpersonen und die fehlende Community die Entscheidung gegen CoCo/R.

Im Vergleich dazu erscheint JavaCC dank der Unterstützung von Sun wesentlich verlässlicher und angesichts der zahlreichen Features auch anwenderfreundlicher. Die Beschränkung auf Java und insbesondere die Anlehnung an die Java-Syntax in den Grammatiken, durch die diese umfangreicher und schwerer lesbar werden, trüben den positiven Eindruck. Ferner wirkt die Baumerzeugung mit einem zusätzlichen Verarbeitungsschritt umständlich.

Die Ähnlichkeit zwischen ursprünglicher und ANTLR-spezifischer DMIS-Grammatik wird sicherlich am größten ausfallen, und zwar inklusive der zum Aufbau eines Baumes nötigen Erweiterungen. Das Verhalten von ANTLR, eine

1 Vgl. Parr, T., ANTLR Reference Manual, 2003, Abschnitt „ANTLR Meta-Language“

2 Vgl. Parr, T., ANTLR Reference Manual, 2003, Abschnitt „Grammar annotations for building ASTs“

bestimmte Annahme zu treffen, wenn keine anders lautenden Angaben vorhanden sind, unterstützt das Ziel, die erforderlichen Kenntnisse im Compilerbau gering zu halten. Der DMIS-Checker soll also einen von ANTLR generierten Parser nutzen. Für den Grammatik-Generator soll entsprechend auch ANTLR vorgegeben sein, einerseits als Generator des Parsers für die Ausgangs-Grammatik, und andererseits als Format für die zu generierende Grammatik.

2.2.4 Beschreibung von ANTLR

Dieser Abschnitt gibt einen Überblick über die Nutzung und Funktionsweise des Compiler-Generators ANTLR.

2.2.4.1 Grammatiken

Dieser Abschnitt fasst die Beschreibung des Grammatik-Formats aus der ANTLR-Dokumentation[1] zusammen.

Zu Beginn einer Grammatik kann ein Block Quellcode angegeben werden, der in den Kopf der Dateien mit den generierten Klassen übernommen wird. Hier ist Platz für zum Beispiel `import`- oder `package`-Anweisungen. Darauf folgt eine Klassendefinition, die den Namen der zu generierenden Klasse enthält. Außerdem wird hier angegeben, von welcher Klasse diese Klasse abgeleitet werden soll. ANTLR erwartet jedoch, dass es eine entsprechende Grammatik gibt, um seine Grammatik-Vererbung anzuwenden. Die Parser-Klasse muss folglich direkt oder indirekt von der ANTLR-Parser-Oberklasse abgeleitet sein, und es muss zu jeder Klasse dieser Hierarchie eine Grammatik existieren.

Alles nach dieser und vor der nächsten Klassendefinition gehört zu dieser Klasse. Nach einer Klassendefinition kann ein Block mit Optionen für die gerade definierte Klasse stehen, in dem zum Beispiel die Größe des zu verwendenden Lookaheads festgelegt wird. Den größten Teil der Parser- beziehungsweise Scanner-Definition machen jedoch die Regeln aus.

1 Vgl. Parr, T., ANTLR Reference Manual, 2003

Eine Regel besteht aus einem Bezeichner, dem Definitionssymbol „:“ und dem Regelkörper, der durch ein Semikolon abgeschlossen wird:

```
datdef : "DATDEF"^ TS_SLASH! var_datdef;
```

Im obigen Beispiel beginnt der Regelkörper mit dem Terminal `DATDEF`. Das daran anschließende Symbol „^“ bewirkt, dass dieses Terminal zur Wurzel eines Unterbaums wird. Dieser Unterbaum wird den Rest des Regelkörpers enthalten. Als zweites wird die Regel `TS_SLASH` referenziert. Das Ausrufezeichen veranlasst ANTLR, dieses Token nicht in den Baum aufzunehmen. Das dritte Element ist ebenfalls eine Regelreferenz.

Alternativen werden durch einen senkrechten Strich getrennt, und mehrere Elemente durch runde Klammern zu einer Gruppe zusammengefasst. Beliebig häufige Wiederholung wird durch ein Sternchen hinter der schließenden Klammer ausgedrückt. Ist eine Gruppe optional, steht hinter der schließenden Klammer ein Fragezeichen.

Im Regelkörper kann an jeder Stelle Quelltext eingefügt werden. Dieser muss in geschweifte Klammern eingeschlossen werden und wird von ANTLR entsprechend seiner Position in die für diese Regel generierte Methode übernommen. Steht er zum Beispiel hinter einem Terminal, wird er ausgeführt, nachdem dieses Terminal erkannt worden ist.

Für eine Regel oder eine Gruppe können Optionen angegeben werden. Die wichtigste Option ist hier `greedy`. Wird sie auf `true` gesetzt, generiert ANTLR Code, der so viel wie möglich zum Beispiel innerhalb einer Schleife erkennt, bevor aus der Schleife ausgestiegen wird. Dies kann helfen, Mehrdeutigkeitswarnungen zu verhindern. Für den gleichen Zweck können auch syntaktische und semantische Prädikate verwendet werden. Diese sind jedoch weitaus vielseitiger und können zum Beispiel auch zur Simulation von Zuständen verwendet werden.

Die Regelnamen im Parser müssen mit einem kleinen Buchstaben beginnen, die im Lexer mit einem großen. Ferner dürfen Terminale, die nur aus einem einzelnen Zeichen bestehen, nur im Lexer vorkommen. Im Parser können virtuelle Token verwendet werden, wenn ein Symbol in einer Regel benötigt

wird, das nicht vom Lexer im eigentlichen Sinne erkannt wird. Das Symbol für das Dateiende ist zum Beispiel ein solches virtuelles oder gedachtes Token. Ein weiterer Anwendungsfall ist, wenn im Lexer während der Erkennung eines Tokens dessen Typ durch eine semantische Aktion geändert wird. Der neue Typ kann ein virtuelles Token sein. Beispiele hierfür werden im Abschnitt Implementierung vorgestellt. Alle in einer Grammatik verwendeten virtuellen Token müssen im Block `tokens` angegeben werden.

2.2.4.2 Anwendung und Arbeitsweise der generierten Klassen

Zunächst ist eine Instanz des Lexers zu erzeugen. Dabei wird eine Referenz auf eine Instanz der Klasse `InputStream` übergeben, die einen Zeichenstrom aus der Eingabedatei liefert. Bei der Instanzierung des Parsers wird die Referenz des Lexers übergeben. Anschließend kann durch Aufruf einer Methode, die den Namen einer Regel aus der Grammatik trägt, das Parsen gestartet werden. Es können also auch Teile einer Datei geparst werden, wenn eine passende Methode aufgerufen wird. Um zum Beispiel eine vollständige DMO-Datei zu parsen, müsste die Methode `output_file()` aufgerufen werden. Beim Entwurf muss berücksichtigt werden, dass der Name der Methode zum Starten des Parsens von der Grammatik abhängt.

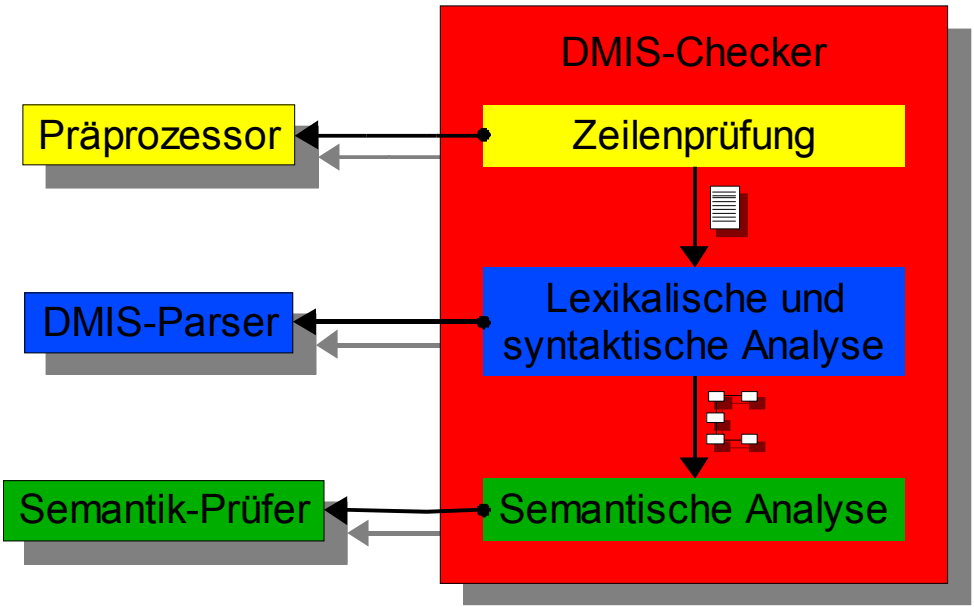
Die Methode `LA(int k)` liefert den Typ des k-ten Tokens. Sie wird in den Methoden, die den Regeln der Grammatik entsprechen, benutzt, um den Typ der nächsten Token aus dem Token-Strom zu ermitteln. Anhand dieser Token wird bei Wahlmöglichkeiten entschieden, welcher Weg gegangen wird. Der bei der Generierung des Parsers angegebene Lookahead stellt die Obergrenze dar, wie viele Token der von ANTLR generierte Quellcode zur Entscheidung heranzieht. Reichen die *k* zur Verfügung stehenden Token nicht aus, um eine Entscheidung zu treffen, liegt Nichtdeterminismus vor. In diesem Fall wählt ANTLR die erste der Alternativen. Außerdem gibt ANTLR bei der Generierung des Parsers eine entsprechende Warnung aus. Ein Aufruf der Methode `LA(int k)` bewirkt letztendlich einen Aufruf der Methode `nextToken()` des Lexers. Diese ist quasi die Startregel für den Lexer. Der Lexer arbeitet prinzipiell wie der Parser, aber mit Zeichen anstatt mit Token. Wird an einer bestimmten Stelle der Eingabedatei ein bestimmtes Token oder

Zeichen erwartet, wird die Methode `match()` aufgerufen. Diese überprüft, ob das erwartete Token oder Zeichen vorliegt und konsumiert es gegebenenfalls. Andernfalls wird eine Ausnahme geworfen. Beim Konsumieren wird das aktuelle Zeichen an das Lexem des Tokens, das gerade erkannt wird, angehängt, beziehungsweise das aktuelle Token in den Baum aufgenommen, wenn einer erzeugt werden soll.

Um eine korrekte Zeilennummer zu gewährleisten, muss nach jedem erkannten Zeilenumbruch die Methode `newline()` aufgerufen werden.

2.3 DMIS-Checker

Der DMIS-Checker ist in verschiedene Phasen unterteilt, die aufeinander aufbauen. Diese Struktur veranschaulicht Darstellung 10. In jeder dieser Phasen werden bestimmte Prüfungen durchgeführt und eine andere Darstellung der Eingabe-Datei erzeugt. Die lexikalische und syntaktische Analyse erhält von der Zeilenprüfung eine veränderte Textdatei und gibt eine Baumdarstellung an die semantische Analyse weiter.



Quelle: Eigene Darstellung

Darstellung 10: Aufbau des DMIS-Checkers

Der DMIS-Checker steuert lediglich den Ablauf der Prüfung. Die eigentliche Arbeit delegiert er an andere Klassen. Diese Modularisierung sorgt für die nötige Flexibilität, um eine Unterstützung von verschiedenen DMIS-Versionen und -Formaten zu ermöglichen. Andernfalls würde jede neue Version eine Änderung im Quelltext erfordern. Außerdem wäre die Wiederverwendbarkeit des Parsers nicht gegeben.

Um den DMIS-Checker leicht in andere Anwendungen integrieren zu können, soll die eigentliche Prüfung nicht in der `main()`-Methode erfolgen, sondern in einer eigenen öffentlichen Methode, die von der `main()`-Methode mit den Parametern der Kommandozeile aufgerufen wird.

Da verschiedene Versionen von DMIS geprüft werden sollen, muss bei Anwendung des DMIS-Checkers angegeben werden, auf Einhaltung welches Formats und welcher DMIS-Version eine Datei getestet werden soll. Anhand dieser Informationen müssen die richtigen Prüfwerkzeuge, nämlich Präprozessor, Parser und Semantik-Prüfer ermittelt werden.

2.3.1 Phasen des DMIS-Checkers

Der DMIS-Checker stellt im Prinzip einen halben Compiler dar, denn er muss alle Phasen des Analyseteils durchführen. Der Syntheseteil fällt hingegen weg.

2.3.1.1 Zeilenprüfung

Bevor die Analyse im Sinne des Compilerbaus beginnt, muss die zu prüfende Datei von einem Präprozessor verarbeitet werden, da in DMIS an jeder Stelle Zeilenumbrüche erlaubt sind. Außerdem gibt es eine Beschränkung der Zeilenlänge auf 80 Zeichen.[1] Diese Gegebenheiten sind jedoch in der Grammatik nicht berücksichtigt, und der Aufwand, sie entsprechend zu ändern, steht in keinem Verhältnis zum Nutzen. Die Grammatik würde wesentlich komplizierter und unleserlicher. Daher wird ein Präprozessor vorangestellt, der die Zeilenlängen prüft und Zeilenumbrüche innerhalb einer Anweisung entfernt. Anschließend entspricht das Zeilenende dem Anweisungsende, und innerhalb einer Anweisung kommt kein Zeilenumbruch vor. Das ermöglicht dem Parser ein Wiederaufsetzen, nachdem ein Fehler in einer Anweisung gefunden wurde, weil das Ende der Anweisung leicht gefunden werden kann. Dadurch können die restlichen zur fehlerhaften Anweisung gehörigen Zeichen ignoriert und die Prüfung bei der nächsten Anweisung fortgesetzt werden.

2.3.1.2 Lexikalische und syntaktische Analyse

Die lexikalische Analyse übernimmt der Lexer, die syntaktische Analyse der Parser. Diese sind eng miteinander verzahnt, so dass der DMIS-Checker mit dem Lexer nichts zu tun hat. Die lexikalische Analyse wird vom Lexer je nach Bedarf des Parsers durchgeführt.

1 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 19

Die Grenze zur semantischen Analyse ist fließend, denn die syntaktische Analyse kann auch einige semantische Fehler finden, wenn entsprechende semantische Informationen in Regeln der Grammatik eingeflossen sind. Dies ist bei der DMIS-Grammatik der Fall. So ist zum Beispiel der Aufbau jedes einzelnen in DMIS vorhandenen Befehls beschrieben. Es werden sogar alle für einen Befehl zulässigen Parameter-Kombinationen und bei den einzelnen Parametern der zulässige Datentyp und Wertebereich angegeben. Das ist nur möglich, weil zur Unterscheidung, zum Beispiel von Wertebereichen, das Lexem, also der Text eines Tokens, ausreicht. Außerdem erleichtert DMIS die Unterscheidung von Datentypen bei Labels, weil sich diese aus einem Typkürzel und dem eigentlichen Bezeichner zusammensetzen. Die semantische Information über den Typ eines Labels wird somit durch das Kürzel zu einem Teil der Syntax. Folglich wäre die Typunterscheidung bei Labels ohne das Typkürzel nur in der semantischen Analyse möglich.

2.3.1.3 Semantische Analyse

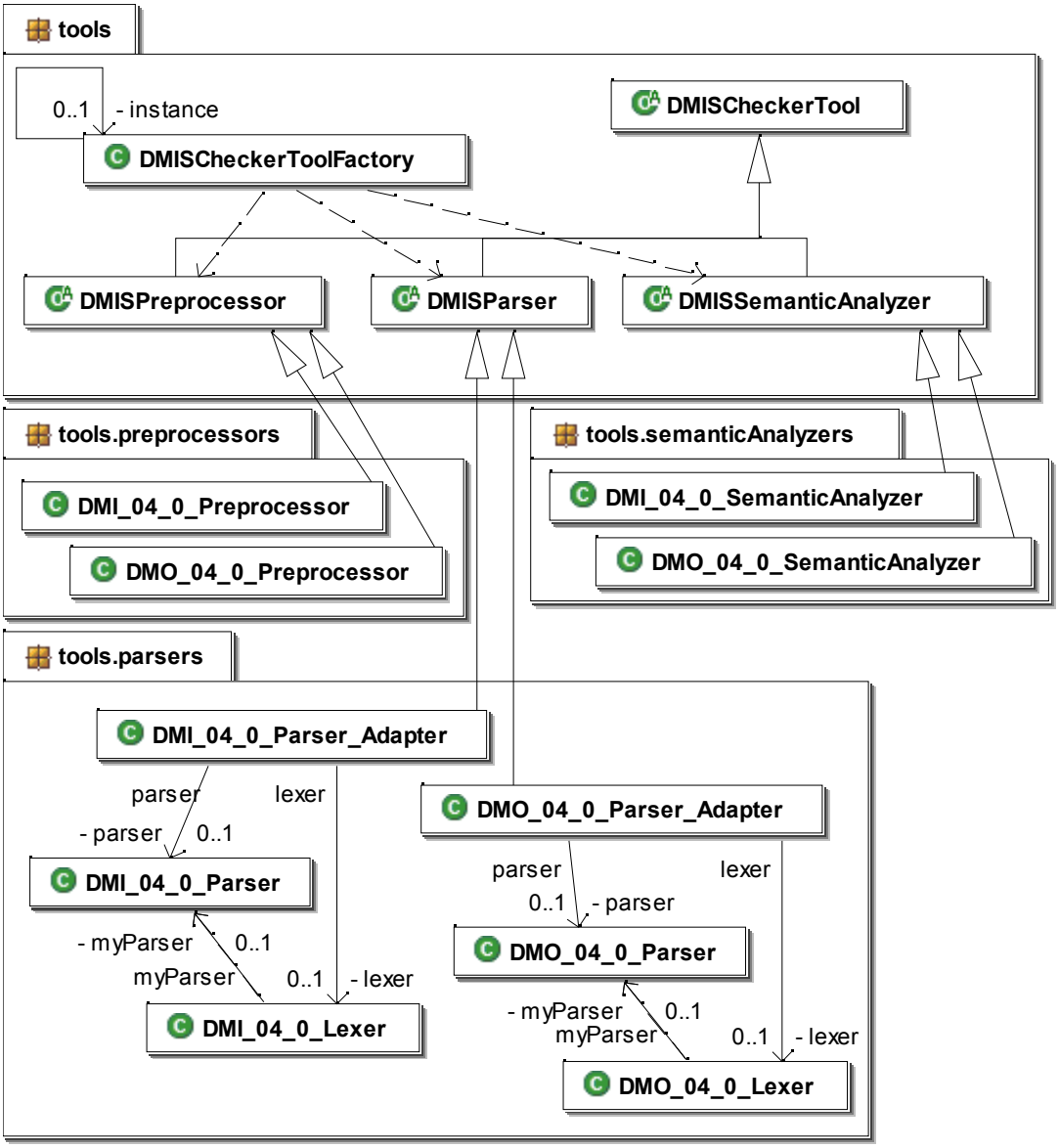
Für die semantische Prüfung ist ein Modul namens Semantik-Prüfer vorgesehen. Die einzelnen Prüfungen müssen manuell implementiert werden, da eine maschinenlesbare Beschreibung der Semantik von DMIS fehlt. Sie wird durch den Standard nur textuell beschrieben. Dies hat Mehrdeutigkeiten und Ungenauigkeiten zur Folge. Aber auch die Existenz einer formalen Beschreibung würde eine Generierung von semantischen Prüfungen nicht unbedingt ermöglichen, weil das Gebiet der Semantik-gesteuerten Compiler noch nicht so weit fortgeschritten ist.

Es wird jeweils nur eine semantische Prüfung implementiert werden, die als Muster für zukünftige Erweiterungen dienen soll. Beim DMO-Format wird die OUTPUT-Anweisung daraufhin geprüft werden, ob alle Parameter ausgewertet wurden und entsprechende Ausgaben erfolgt sind. Beim DMI-Format wird geprüft werden, ob eine Variable deklariert wurde, bevor sie benutzt wird.

2.3.2 Architektur der Prüfwerkzeuge

Damit der DMIS-Checker Prüfwerkzeuge für verschiedene DMIS-Versionen und -Formate gleichzeitig bereithalten kann, und das für eine zu prüfende

Datei jeweils richtige gefunden werden kann, müssen sowohl Format als auch Version im Namen der Klasse stehen. Diese Klassen müssen jeweils eine gemeinsame Oberklasse besitzen, die sicherstellt, dass jedes Prüfwerkzeug eine festgelegte Methode zur Verfügung stellt, über die die jeweilige Prüfung gestartet werden kann. Es ist nicht nötig und auch nicht wünschenswert, dass der DMIS-Checker die verschiedenen Versionen der einzelnen Prüfwerkzeuge kennt, sondern er muss mit den Oberklassen arbeiten. Dadurch kann ein neu entwickeltes Prüfwerkzeug sofort ohne Änderungen am DMIS-Checker genutzt werden.



Quelle: Eigene Darstellung

Darstellung 11: Klassendiagramm Prüfwerkzeuge

Darstellung 11 zeigt die Beziehungen zwischen den Klassen der Prüfwerkzeuge. Bei den konkreten Prüfwerkzeugen sind hier beispielhaft nur die Klassen für Ein- und Ausgabe-Format von DMIS 04.0 aufgeführt. Diese sind in die Pakete `tools.preprocessor`, `tools.parsers` und `tools.semanticAnalyzers` aufgeteilt.

Die abstrakte Klasse `DMISPreprocessor` ist beispielsweise die Oberklasse für alle Präprozessoren, entsprechend `DMISSemanticAnalyzer` für alle Semantik-Prüfer.

Für die Parser ist diese Architektur jedoch so nicht umsetzbar, da die `extends`-Klausel der Parser-Definition in einer ANTLR-Grammatik nicht die gleiche Bedeutung hat wie in einer Klassendefinition in Java. ANTLR bezieht die Klausel auf die Grammatik, nicht auf die Parser-Klasse, das heißt, es muss eine Grammatik mit dem entsprechenden Namen geben. Die `extends`-Klausel der generierten Parser-Klasse bleibt von der Angabe unberührt.[1] Um dennoch eine gemeinsame Oberklasse zu ermöglichen, kommt das Adapter-Muster zur Anwendung.[2]

`DMI_04_0_Parser_Adapter` kapselt zum Beispiel eine Instanz der von ANTLR generierten Klasse `DMI_04_0_Parser` und erbt von der abstrakten Klasse `DMISParser`. Analog verhält es sich mit dem Parser für das Output-Format. Für eine weitere DMIS-Version kämen je Format zwei Klassen hinzu, nämlich eine von ANTLR generierte und eine Adapter-Klasse.

Die Adapter-Klassen haben noch zwei weitere Vorteile: Dadurch, dass diese in der Methode `doChecking()` den Aufruf der Methode zum Starten des Parsens übernehmen, lösen sie nebenbei das Problem der variierenden Methodennamen. Denn der Name der Methode, die zum Starten des Parsens aufgerufen werden muss, hängt von der Grammatik ab, aus der der Parser generiert wurde. Außerdem verbergen die Adapter-Klassen die Details der Parser-Anwendung, so dass der DMIS-Checker diesbezüglich Parser-Generator unabhängig ist. Diese Unabhängigkeit reicht allerdings nur bis zur Übergabe des Baums an den Semantik-Prüfer.

---

1 Vgl. Parr, T., ANTLR Reference Manual, 2003, Abschnitt „Grammar Inheritance“

2 Vgl. Gamma, E., Entwurfsmuster, 1996, S. 171

Die Werkzeug-Oberklassen `DMISPreprocessor`, `DMISParser` und `DMISSemanticAnalyzer` sind von der abstrakten Klasse `DMISCheckerTool` abgeleitet. Diese legt den Namen der Methode zum Prüfen und deren Ablauf fest. Dies nennt *Gamma* eine Schablonenmethode, das heißt, die Methode setzt sich aus mehreren Methodenaufrufen zusammen. Diese aufgerufenen Methoden können beziehungsweise müssen von den erbenden Klassen überschrieben werden. Dadurch können Teile des Algorithmus geändert werden, nicht aber sein grundsätzlicher Aufbau.[1]

Eine Prüfung soll damit beginnen, dass zuerst die Version und der Name desjenigen Prüfwerkzeuges ausgegeben werden, das im Begriff ist, eine Prüfung durchzuführen. Danach soll geprüft werden, ob alle Voraussetzungen zum Start der Prüfung gegeben sind. Insbesondere soll sicher gestellt werden, dass die zu prüfenden Daten vorliegen. Abschließend erfolgt die eigentliche Prüfung.

Die Instanzen der konkreten Prüfwerkzeuge für eine DMIS-Version und ein DMIS-Format erstellt die Fabrik-Klasse `DMISCheckerToolFactory`. Diese stellt unter anderem eine Klassenmethode zur Verfügung, die aus Format und Version den Namen der entsprechenden Klasse erzeugt. Dieser soll mit dem Kürzel DMI oder DMO für das Format beginnen. Darauf folgen die Haupt- und Nebenversionsnummer mit zwei beziehungsweise einer Stelle. Beim Präprozessor steht anschließend „Preprocessor“, beim Parser „Parser“ und beim Semantik-Prüfer „SemanticAnalyzer“. Bei einer Adapter-Klasse wird „Adapter“ angehängt. Die einzelnen Bestandteile des Namens trennt ein Unterstrich.

2.3.3 Baumdarstellung einer DMIS-Datei

Die semantische Analyse wird umso mächtiger, je besser der zugrunde liegende Baum strukturiert ist. Die Struktur entsteht durch die Entscheidung, wann ein Unterbaum erstellt wird. Enthält ein Baum keine Unterbäume, ist er zu einer verketteten Liste entartet. Da jedoch der Baum von dem möglichst automatisch generierten Parser aufgebaut wird, ergibt sich ein Konflikt zwischen der einfachen Erzeugung des Parsers durch den Grammatik-

1 Vgl. Gamma, E., Entwurfsmuster, 1996, S. 366

Generator und dem Grad der Strukturierung des Baumes. Eine vollautomatische Identifizierung von geeigneten Unterbäumen ist keinesfalls möglich, da hierzu semantische Informationen nötig sind, nämlich die Kenntnis der Bedeutung der einzelnen Befehle und ihrer Parameter, über die der Grammatik-Generator nicht verfügt.

Der Aufbau der DMIS-Anweisungen erlaubt allerdings eine relativ einfache Aufteilung in Teilbäume, bei der jeder Teilbaum genau eine Anweisung darstellt. Folgendem Muster folgen alle DMIS-Anweisungen[1]:

```
MAJOR_WORD / MINOR_WORD(S), PARAMETER(S)      oder
LABEL = MAJOR_WORD / MINOR_WORD(S), PARAMETER(S)
```

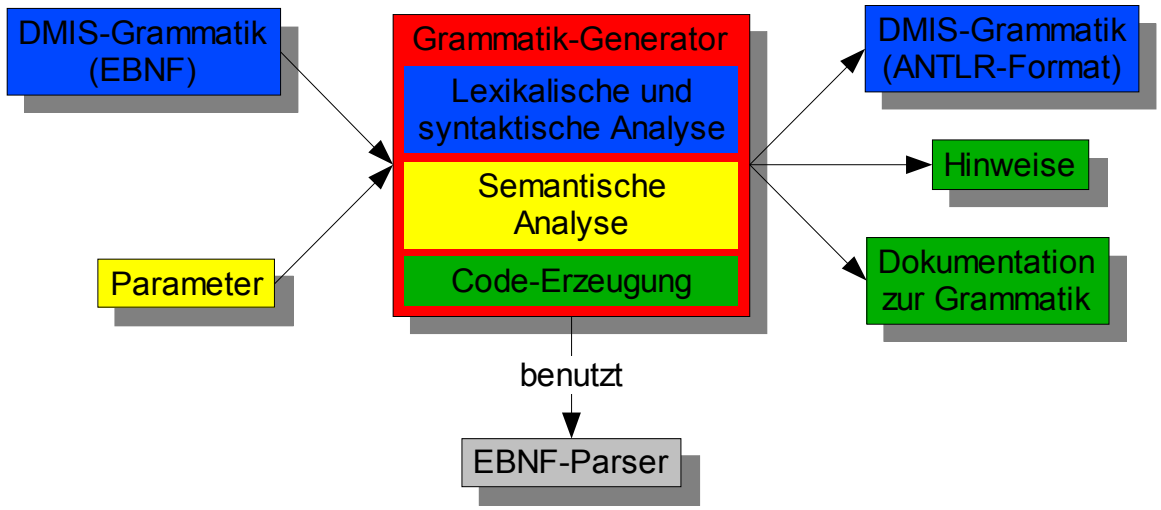
Im ersten Fall wird das Major-Word zur Wurzel des Unterbaums, im zweiten das Label. Da Label jedoch auch als Parameter auftreten können, muss hier ein zusätzliches Merkmal her. Die Kombination von Label und Gleichheitszeichen würde in diesem Fall ausreichen. Die Problematik der Identifizierung von Unterbäumen wird im Zusammenhang mit dem Grammatik-Generator weiter behandelt.

Auch wenn die semantische Analyse jeden Fehler findet, ist sie nutzlos, wenn die Position der Fehler nicht angegeben werden kann. Daher muss diese Information beim Aufbau des Baumes in die Knoten übernommen werden, wozu entsprechende Felder in den Knoten nötig sind. Für diesen Zweck wurde die Klasse `RichAST` entworfen, die für alle Knoten verwendet wird.

1 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 16

2.4 Grammatik-Generator

Die DMIS-Grammatik im EBNF-Format ist die Eingabe für den Grammatik-Generator. Mit weiteren Parametern soll die Verarbeitung gesteuert werden können. Als Ergebnis liefert der Generator eine Grammatik im ANTLR-spezifischen Format und Hinweise auf mögliche Probleme in der Grammatik. Die Erzeugung von Dokumentation zur Grammatik soll über entsprechende Parameter angestoßen werden können. Einen Überblick über den Grammatik-Generator gibt Darstellung 12.



Quelle: Eigene Darstellung

Darstellung 12: Aufbau des Grammatik-Generators

Der Grammatik-Generator ist vom Aufbau her ein Compiler, auch wenn sich die Übersetzung im Wesentlichen auf ein Umformatieren beschränkt. Die lexikalische und syntaktische Analyse bedient sich eines EBNF-Parsers und erzeugt eine Baumdarstellung der Eingabe-Grammatik. Die semantische Analyse führt an Hand des Baumes verschiedene Prüfungen und Änderungen durch. Aus dem bearbeiteten Baum wird schließlich die Grammatik im ANTLR-Format erzeugt.

2.4.1 DMIS-Grammatik im EBNF-Format

Die Regeln der DMIS-Grammatik setzen sich aus einem Regelbezeichner, dem Definitionssymbol und der Regeldefinition zusammen. Im folgenden Beispiel

ist <tecomp> der Regelbezeichner und „::=“ das Definitionssymbol.

```
<tecomp> ::= TECOMP '/' <var_tecomp>
```

Rechts vom Definitionssymbol steht die Regeldefinition. Sie beginnt mit den beiden Terminalen TECOMP und /. Darauf folgt eine Referenz der Regel <var_tecomp>. Terminale, die nur ein Zeichen lang sind, müssen in Hochkommata eingeschlossen werden, andernfalls müssen sie nicht gekennzeichnet werden. Um Regelbezeichner von Terminalen zu unterscheiden, sind diese in spitze Klammern einzuschließen. Ob ein Regelbezeichner als Beginn einer neuen Regel oder als Referenz auf die entsprechende Regel gemeint ist, wird aus seiner Position deutlich. Nur wenn er unmittelbar vor einem Definitionssymbol steht, handelt es sich um den Beginn einer neuen Regel.

Es folgt die im obigen Beispiel referenzierte Regel:

```
<var_tecomp> ::= ON [ ',' <real> ] | OFF
```

Der senkrechte Strich vor dem Terminal OFF ist ein Oder-Operator, das heißt, diese Regel bietet eine Wahlmöglichkeit zwischen zwei Alternativen. Die erste Alternative besteht aus dem Terminal ON und zwei in eckige Klammern gefasste Elemente. Die eckigen Klammern kennzeichnen eine optionale Gruppe.

Die Regel <Label_char_list> zeigt eine Wiederholungs-Gruppe:

```
<Label_char_list> ::= <Label_char> [ <Label_char> ...63]
```

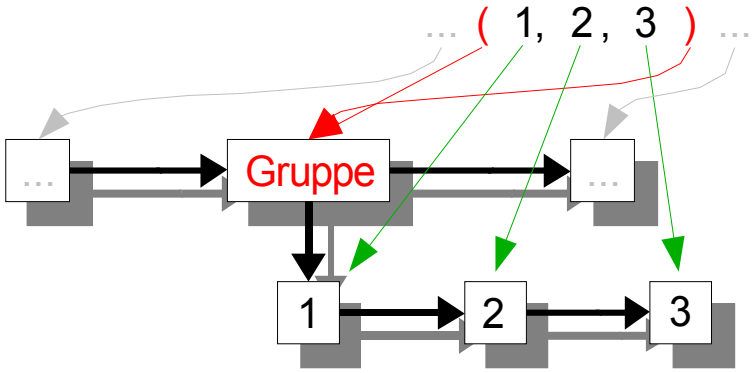
Die Wiederholungs-Gruppe wird ebenfalls durch eckige Klammern gekennzeichnet. Vor der schließenden Klammer stehen jedoch drei Punkte. In diesem Beispiel ist darüber hinaus noch eine Obergrenze für die Anzahl der Wiederholungen angegeben, was aber nicht zwingend ist. An der Regel ist außerdem zu sehen, wie ein- oder mehrmaliges Vorkommen formuliert werden kann.

2.4.2 Baumdarstellung der Grammatik

Die zentrale Datenstruktur für die Arbeit des Grammatik-Generators ist der Baum, denn er enthält nach dem Parsen die gesamten Informationen der Grammatik. Jede Operation wird hierauf ausgeführt. Deswegen sollte der Baum leicht handhabbar sein. Dies entscheidet sich bei seinen Knoten und seiner Struktur.

Zur Formulierung der Regeln einer Grammatik im EBNF-Format werden verschiedene Konstrukte benutzt. Das wichtigste, aber auch das einfachste Konstrukt ist das Terminal, weil es beim Parsen darum geht, zu einem Zeichen oder einer Zeichenkette aus der Eingabe ein Terminal mit dem gleichen Text zu finden. Alle anderen Konstrukte dienen der Strukturierung der Terminale. Folglich macht eine Gruppe ohne Inhalt keinen Sinn. Jedes Terminal wird durch einen eigenen Knoten im Baum dargestellt.

Um kenntlich zu machen, welche Knoten zusammen gehören, zum Beispiel die Elemente einer optionalen Gruppe oder Wiederholung, wird ein Knoten erzeugt, der das Konstrukt „Gruppe“ repräsentiert. Diesem Knoten werden die Elemente der entsprechenden Gruppe untergeordnet.

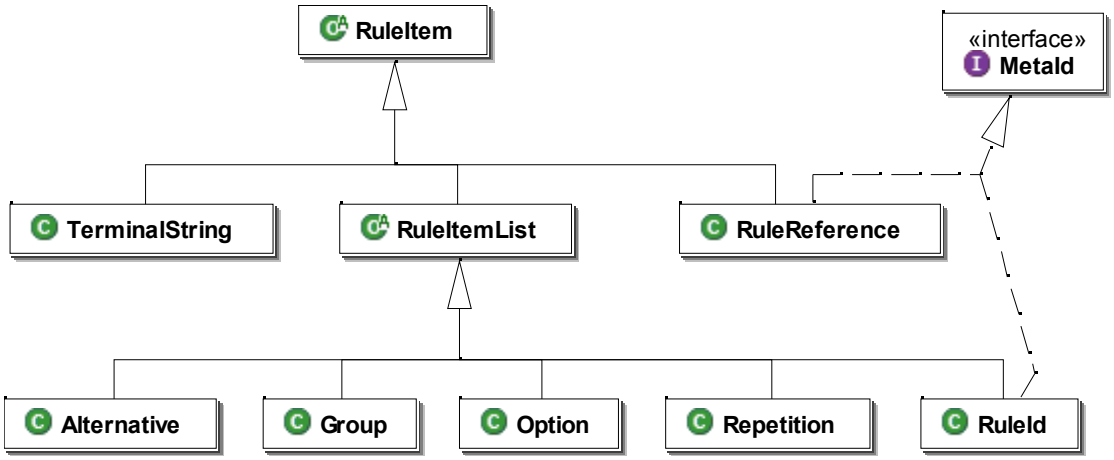


Quelle: Eigene Darstellung

Darstellung 13: Syntaktisches Konstrukt und zugehörige Baumdarstellung

Einen solchen Teil eines Baumes zeigt schematisch Darstellung 13. Die Klammern werden zum Knoten „Gruppe“, die drei Elemente innerhalb der Klammern zu untergeordneten Knoten. Eine Gruppe kann auch als Startpunkt einer Liste von Elementen betrachtet werden, die zusätzliche Merkmale

besitzen kann. Zum Beispiel kann bei einer Wiederholungs-Gruppe eine Obergrenze für die Anzahl der Wiederholungen angegeben werden. Aber auch wenn keine zusätzlichen Angaben gespeichert werden müssen, muss eine eigene Klasse verwendet werden. Eine Gruppe und eine optionale Gruppe unterscheiden sich beispielsweise nur in ihrer Semantik.



Quelle: Eigene Darstellung

Darstellung 14: Klassenhierarchie für die Knoten des AST im Grammatik-Generator

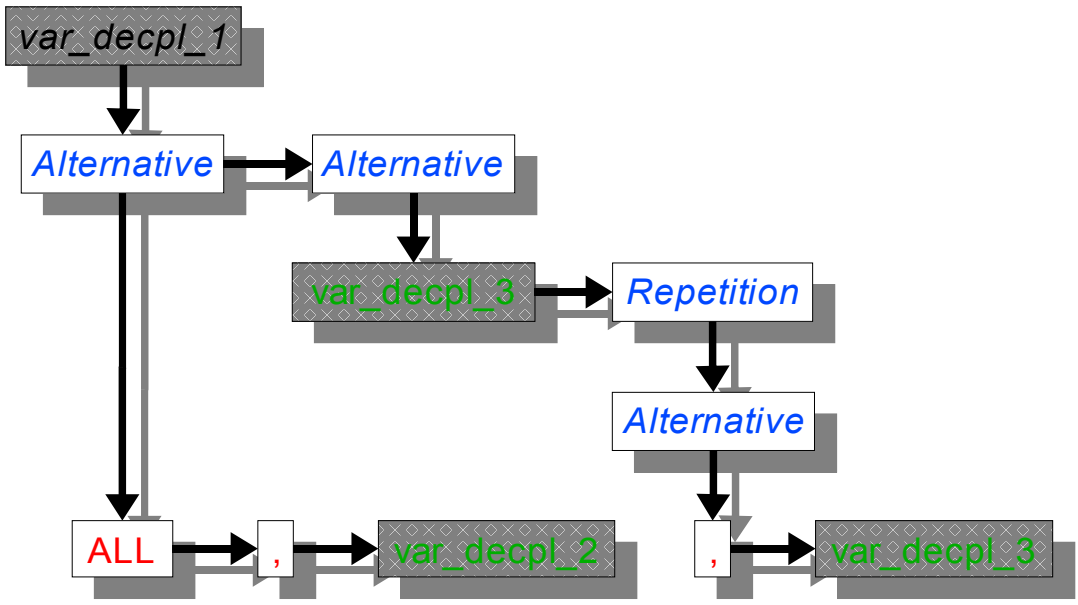
Unabhängig davon, was er repräsentiert, soll jeder Knoten vom Typ `RuleItem` sein. Daher steht dieser Typ an oberster Stelle der in Darstellung 14 gezeigten Klassenhierarchie. Auf der nächsten Ebene wird zwischen Terminalen und Nonterminalen, also Listen allgemein und Regelreferenzen, unterschieden. Eine konkrete Liste ist eine Alternative, Gruppe, optionale Gruppe, Wiederholungs-Gruppe oder ein Regelname, also die Wurzel einer Regel. Die Klassen `RuleId` und `RuleReference` implementieren zusätzlich das Interface `MetaId`. Dadurch können mittels einer Prüfung auf diesen Typ allgemein Regelnamen, gleich an welcher Stelle einer Regel sie stehen, aufgefunden werden.

Die Klassen `RuleItem` und `RuleItemList` sind abstrakt und werden die meiste Funktionalität der Knoten zur Verfügung stellen. `RuleItem` ist von der Klasse `CommonAST` aus dem Paket `antlr` abgeleitet, damit die Knoten von ANTLR im Baum verwendet werden können. Außerdem implementiert `RuleItem` das Interface `Cloneable`, wodurch Manipulationen am Baum vereinfacht werden sollen.

Dass zum Beispiel bei einer Wiederholung innerhalb der Klammern eine Liste von Elementen steht, ist leicht einzusehen. Dass aber die Gruppe eine Liste von Listen sein muss, wird erst dann deutlich, wenn innerhalb der Gruppe der Oder-Operator vorkommt. Damit der Operator nicht verloren geht, müssen die Elemente der Alternativen jeweils als Liste in den Baum aufgenommen werden, und die Listen-Wurzeln müssen von einem entsprechenden Typ sein. Eine weitere Liste nur dann einzufügen, wenn tatsächlich mindestens einmal der Oder-Operator auftritt, würde das Durchwandern des Baumes erschweren. Daher sollen alle Instanzen der Klassen `RuleId`, `Group`, `Option` und `Repetition` nur Instanzen der Klasse `Alternative` als Kinder haben können. Umgekehrt sollen Knoten der Klasse `Alternative` alle Knoten außer denen des Typs `Alternative` als Kinder haben können.

Als Beispiel für die Verwendung der verschiedenen Knoten dient die folgende EBNF-notierte Regel:

```
<var_decpl_1> ::=
ALL ',' <var_decpl_2> | <var_decpl_3> [ ',' <var_decpl_3> ...]
```



Quelle: Eigene Darstellung

Darstellung 15: Baumdarstellung der Regel "var_decpl_1"

Die Baumdarstellung dieser Regel zeigt Darstellung 15. Instanzen der Klasse `TerminalString` sind in roter, der Klasse `RuleReference` in grüner und der Klasse `RuleId` in schwarzer Schrift gezeichnet. Die durch schraffierte Kästchen dargestellten Knoten sind auch vom Typ `MetaId`. Kursive Schrift lässt den Typ `RuleItemList` erkennen. Bei blauer Schrift steht der Name der jeweils instanzierten Klasse im Kästchen.

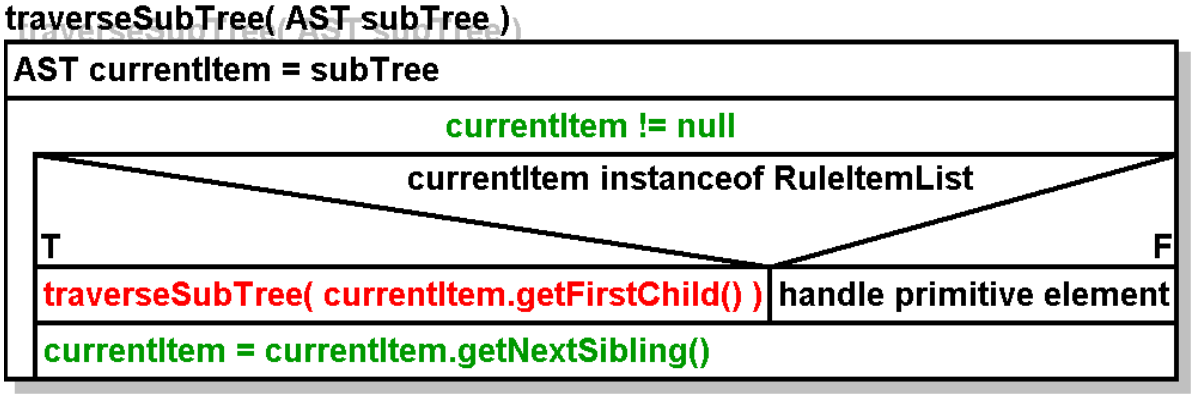
Es ist auch zu sehen, wie Alternativen verwendet werden. Der `RuleId`-Knoten „var_decpl_1“ hat eine Liste von Alternativen, die zwei Elemente umfasst. Diese Alternativen haben ebenfalls Listen. Die zweite Alternative hat als zweiten Kind-Knoten eine Wiederholungs-Gruppe, die wiederum eine Liste von Alternativen besitzt. Diese Liste enthält allerdings nur ein Element.

3 Implementierung

Da in vielen Methoden mit Bäumen gearbeitet wird, beginnt dieses Kapitel mit einer Erläuterung des grundsätzlichen Aufbaus eines Algorithmus, der einen Baum durchläuft. Anschließend werden die Besonderheiten der Implementierung des DMIS-Checkers und des Grammatik-Generators dargestellt. Dabei wird auch kurz auf die Anwendung des Grammatik-Generators zur Erzeugung der Parser für den DMIS-Checker eingegangen. Eine ausführliche Besprechung würde jedoch den Rahmen dieser Arbeit sprengen.

3.1 Rekursives Durchwandern eines Teilbaumes

Sollen von einer Operation alle Knoten eines Teilbaumes erfasst werden, geht man nach einem einfachen, aber festen Schema vor, um sicherzustellen, dass kein Knoten ausgelassen wird.



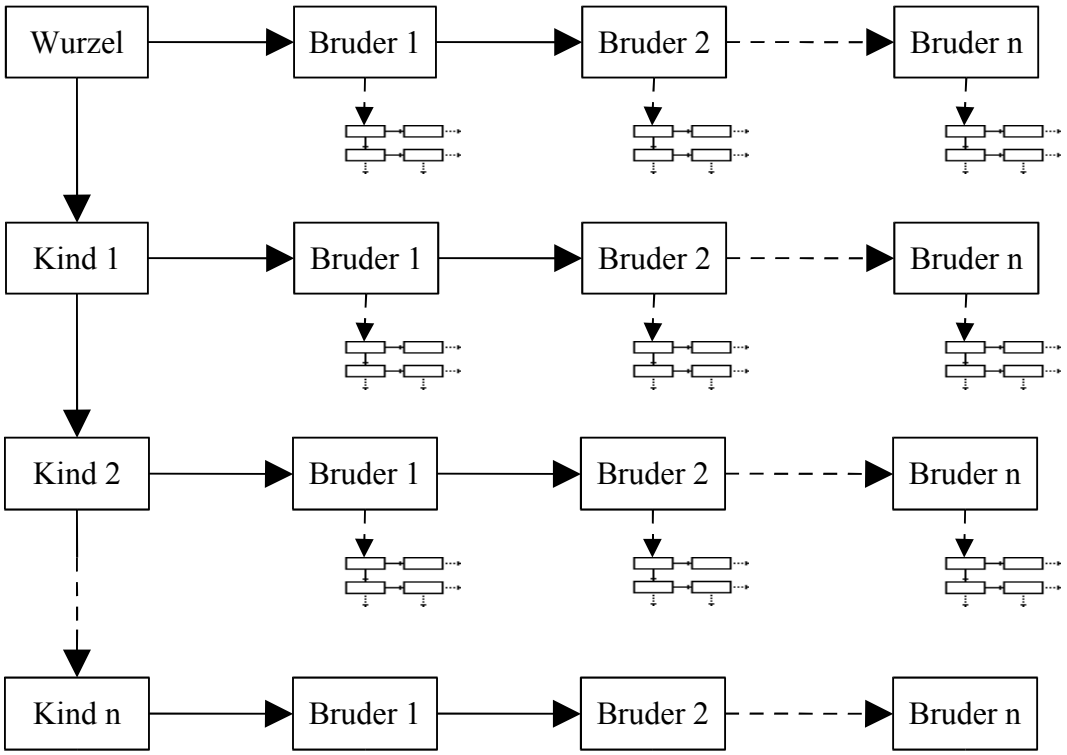
Quelle: Eigene Darstellung

Darstellung 16: Aufbau einer Methode zum Durchwandern eines Baumes

Das Struktogramm in Darstellung 16 zeigt den grundsätzlichen Aufbau einer Methode, die jeden Knoten eines Baumes erreicht. Innerhalb der Methode wird von der übergebenen Wurzel jeweils zum nächsten Geschwister-Knoten gesprungen, während durch rekursiven Aufruf der Methode die Kind-Knoten erreicht werden. Die Behandlung der Geschwister-Knoten endet, wenn es keinen weiteren gibt. Die hierzu nötigen Befehle sind in grün dargestellt. Der rekursive Aufruf der Methode wird durch eine Bedingung gesteuert. Handelt es sich bei dem gerade betrachteten Knoten um eine Instanz der Klasse

`RuleItemList`, besitzt er Kind-Knoten. Daher wird die Methode für das erste Kind des Knotens rekursiv aufgerufen. Anderenfalls handelt es sich um ein einfaches Element, dessen Bearbeitung in der Methode direkt abgeschlossen werden kann. Dadurch endet die Kette rekursiver Aufrufe.

Soll eine Operation nur auf bestimmte Regeln ausgeführt werden, muss der eigentlichen Bearbeitungsmethode, die nach dem obigen Muster aufgebaut ist, eine andere Methode vorangestellt werden. In dieser wird dann entschieden, ob eine bestimmte Regel bearbeitet werden soll und dementsprechend die eigentliche Bearbeitungsmethode aufgerufen. Dieses Vorgehen ist notwendig, weil der gleiche Text an verschiedenen Stellen des Baums verschiedene Bedeutungen haben kann. So stellt der Text des Wurzelknotens einer Regel den Regelnamen dar, während der gleiche Text bei einem Knoten unterhalb des Wurzelknotens eine Referenz auf eine Regel mit diesem Namen meint. Eine Unterscheidung wäre zwar an Hand des Typs der Knoten möglich, würde aber unnötige Komplexität mit sich bringen.



Quelle: Eigene Darstellung

Darstellung 17: Syntaxbaum

Darstellung 17 zeigt den Aufbau eines Baumes, wie er von einem Parser, der von ANTLR generiert wurde, erzeugt wird. Sowohl der Grammatik-Generator als auch der DMIS-Checker verwenden Bäume mit diesem Aufbau. Lediglich die Klassen der Knoten unterscheiden sich.

3.2 DMIS-Checker

Die Klasse `DMIS_Checker` besitzt eine `main()`-Methode, damit sie eigenständig nutzbar ist. Hier werden jedoch lediglich die Kommandozeilenparameter überprüft und an die Methode `check()` weitergereicht. Diese erwartet den Namen der zu prüfenden Datei sowie die DMIS-Version und das -Format, auf die diese geprüft werden soll. Die Methode führt die eigentliche Prüfung durch, indem sie eine Instanz der Klasse `DMISCheckerToolFactory` erzeugt und diese die jeweils benötigten Prüfwerkzeuge instanzieren lässt. Jedes Prüfwerkzeug wird durch geeignete Methodenaufrufe mit den nötigen Parametern versorgt. Anschließend wird die Prüfung durch Aufruf der Methode `startChecking()` angestoßen.

3.2.1 Ablauf der Prüfungen aller Werkzeuge

Die Klasse `DMISCheckerTool` ist Oberklasse aller Prüfwerkzeuge und legt den Ablauf der Prüfung für alle abgeleiteten Klassen fest, indem sie die Methode `startChecking()` implementiert. Die Ausgabe der Informationen zum benutzten Prüfwerkzeug wird durch die Implementierung der Methode `printToolInfo()` ebenfalls festgelegt. Diese nutzt abstrakte Methoden, um die Informationen zu ermitteln. Erbende Klassen müssen Methoden zur Initialisierung (`initChecking()`) und Durchführung (`doChecking()`) der Prüfung und zur Abfrage von statistischen Daten über die Prüfung (`getCntCheckedItems()`, `getCntErrors()`) und Informationen über das Prüfwerkzeug (`getDMISMajorVersion()`, `getDMISMinorVersion()`, `getDMISFormat()`, `getToolName()`) bereitstellen.

3.2.2 Oberklassen der verschiedenen Prüfwerkzeugtypen

Die Klassen `DMISPreprocessor`, `DMISParser` und `DMISSemanticAnalyzer` legen jeweils den Werkzeugnamen fest. Außerdem stellen sie Methoden zur Übergabe und Abfrage der zu prüfenden Daten zur Verfügung und implementieren die Initialisierung vor der Prüfung.

3.2.2.1 Standard-Verhalten von Präprozessoren

Die Oberklasse aller Präprozessoren ist `DMISPreprocessor`. Da eine Prüfung der Zeilenlängen und die Entfernung von Zeilenumbrüchen innerhalb einer

Anweisung bei jedem DMIS-Format und jeder Version erforderlich sind – unter anderem damit die Recovery-Strategie aufgeht – leistet dies im Gegensatz zu den anderen Prüfwerkzeugtypen bereits die Oberklasse. Die zulässige Zeilenlänge wird durch Aufruf der Methode `getMaxLineLength()` ermittelt, so dass die Zeilenlänge bei jeder DMIS-Version angepasst werden kann, indem diese Methode gegebenenfalls durch die konkrete Präprozessor-Klasse überschrieben wird.

Des Weiteren werden Methoden zur Übergabe (`setFileName()`) und Abfrage (`getFileName()`) des Dateinamens der zu prüfenden Datei zur Verfügung gestellt. Bevor die Verarbeitung der Zeilen beginnen kann, muss ein Dateiname übergeben worden sein. Dies wird durch die Implementierung der Methode `initChecking()` sicher gestellt.

3.2.2.2 Standard-Verhalten von Parsern

Um das Parsen anzustoßen, muss der Name der Startregel der zu Grunde liegenden Grammatik bekannt sein. Das ist in der Parser-Oberklasse `DMISParser` nicht der Fall, sondern erst in der konkreten Adapter-Klasse für eine bestimmte DMIS-Version und ein DMIS-Format. Daher kann die Prüfung hier nicht implementiert werden. Die Initialisierung wird dennoch umgesetzt. Anstatt die Variablen zum Beispiel für die Anzahl der gefunden Fehler direkt zurückzusetzen, wird eine abstrakte Methode (`resetCounters()`) aufgerufen. Ferner stellt die abstrakte Methode `getAST()` sicher, dass die konkreten Parser-Adapter-Klassen eine Methode bereitstellen, über die auf den während des Parsens aufgebauten Baum zugegriffen werden kann.

3.2.2.3 Standard-Verhalten von Semantik-Prüfern

Da für semantische Prüfungen die Baumdarstellung der zu prüfenden Datei benötigt wird, und diese stark vom Parser abhängt, sollen keine semantischen Prüfungen allgemein gültig implementiert werden. Stattdessen wird eine Meldung ausgegeben, dass keine Prüfungen durchgeführt werden, die automatisch verschwindet, wenn eine Unterklasse die Methode `doChecking()` überschreibt.

Die Initialisierung vor einer Prüfung und die Methoden zur Abfrage der statistischen Informationen setzt die Klasse `DMISSemanticAnalyzer` bereits um.

3.2.3 Prüfwerkzeuge für eine bestimmte DMIS-Version

Soll eine bestimmte DMIS-Version geprüft werden, müssen Klassen für alle Prüfwerkzeugtypen erstellt werden, gleich ob deren Verhalten von dem anderer Versionen abweicht. Die Namen der Klassen müssen den Regeln für die Namensgebung folgen, damit die einzelnen Prüfwerkzeuge instanziert werden können. Jede der Klassen muss die von der Oberklasse `DMISCheckerTool` eingeführten abstrakten Methoden zur Ermittlung von DMIS-Version und -Format implementieren.

3.2.3.1 Präprozessor-Klassen

An den Präprozessor-Klassen müssen nur dann Änderungen vorgenommen werden, wenn sich die zulässige Länge einer Zeile geändert hat. Zur Anpassung reicht es aus, die Methode `getMaxLineLength()` zu überschreiben.

3.2.3.2 Parser-Adapter-Klassen

Wichtigste Aufgabe dieser Klassen ist die Implementierung des Startens der Prüfung in der Methode `doChecking()`. Außerdem sind alle übrigen abstrakten Methoden der Oberklassen bezüglich der statistischen Informationen (`resetCounters()`, `getCntCheckedItems()`, `getCntErrors()`) und der Baumdarstellung (`getAST()`) bereitzustellen.

3.2.3.3 Semantik-Prüfer-Klassen

Hier ist über die Implementierung der abstrakten Methoden hinaus nicht zwingend etwas zu tun. Es können jedoch beliebig viele semantische Prüfungen ergänzt werden, um insgesamt den DMIS-Checker aufzuwerten.

3.2.4 Umgesetzte semantische Prüfungen für DMIS 04.0

Obwohl die Umsetzung von Prüfungen, die nicht auf der Grammatik des Standards basieren, kein Bestandteil der Aufgabenstellung ist, aber die Architektur des DMIS-Checkers auf entsprechende Erweiterungen ausgelegt wurde, sind einzelne semantische Prüfungen umgesetzt worden. Diese können als Basis für weitere Prüfungen genutzt werden.

3.2.4.1 DMIS 04.0 Output: OUTPUT-Statement

Das `OUPTUT`-Statement dient zur Ausgabe der mit einem Label verknüpften Werte. Die auszugebenden Labels werden als Parameter übergeben. Die Anweisung aus der DMI-Datei wird in die DMO-Datei übernommen.[1] Auf diese Anweisung folgen in der Reihenfolge ihrer Parameter die zugehörigen Ausgaben. Für jedes Label wird eine Zeile ausgegeben, die mit dem Label beginnt.

Die Methode `output()` der Klasse `DMO_04_0_SemanticAnalyzer` überprüft, ob für jedes Label, das als Parameter an den `OUTPUT`-Befehl übergeben wurde, eine Ausgabe erfolgt ist. Als Beispiel dient dieser DMO-Auszug:

```
24: OUTPUT/F(X),F(Y),T(X)
25: T(X)=...
```

Das `OUTPUT`-Statement erhält drei Parameter, die Ausgabe ist jedoch nur für das letzte Label erfolgt. Die Ausgaben werden nicht stur für das i-te Label in der i-ten Zeile gesucht. Stattdessen wird das erste Token der auf den Befehl folgenden Zeile mit den Labels vom ersten an verglichen und bei Übereinstimmung die nächste Zeile herangezogen. Dies führt bei obigem Beispiel zu folgenden Fehlermeldungen:

```
error: line 25:1: F(X) expected, T(X) found
error: line 25:1: F(Y) expected, T(X) found
```

Aus den Fehlermeldungen ist abzulesen, dass zwei Ausgaben fehlen, und zwar die der beiden Label `F(X)` und `F(Y)`. Für das Label `T(X)` ist jedoch eine Ausgabe erfolgt.

3.2.4.2 DMIS 04.0 Input: Suche nach nicht deklarierten Variablen

In DMIS müssen Variablen vor ihrer Benutzung mittels eines `DECL`-Statements deklariert werden.[2] Dies zu prüfen ist eine klassische Aufgabe der semantischen Analyse. Dazu muss der gesamte Baum durchlaufen und nach Knoten durchsucht werden, die entweder eine Variablen-Deklaration oder eine Verwendung einer Variablen darstellen. Die Namen deklarierter Variablen

1 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 249 - 251

2 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 136 f.

müssen in einer Symboltabelle gespeichert werden, um beim Fund einer Variablen-Verwendung prüfen zu können, ob die verwendete Variable bekannt ist.

Da die Symboltabelle auch für andere Prüfungen nützlich sein kann, wird diese zunächst durch die Methode `fillSymbolTable()` aufgebaut. Dabei werden weitere Informationen wie zum Beispiel Datentyp und Gültigkeitsbereich gesammelt, die dem `DECL`-Statement zu entnehmen sind. Anschließend untersucht die Methode `variablesDeclared()`, ob nicht deklarierte Variablen verwendet werden. Da jedoch bereits zu Beginn der Ausführung der Methode die komplette Symboltabelle zur Verfügung steht, kann an Hand des Vorhandenseins einer Variablen in dieser nicht mehr sicher gestellt werden, dass sie auch vor ihrer Verwendung deklariert wurde. Stattdessen wird dafür die Zeilennummer der Deklaration herangezogen, weshalb diese ebenfalls in der Symboltabelle hinterlegt werden muss. Das ermöglicht genauere Fehlermeldungen, denn es kann gegebenenfalls darauf hingewiesen werden, dass eine Variable erst nach ihrer Verwendung deklariert wird:

```
error: line 22:1: variable "X" used before declaration (-> line 30)!
error: line 59:1: variable "Y" not declared!
```

3.3 Grammatik-Generator

Der Grammatik-Generator besitzt analog zum DMIS-Checker eine `main()`-Methode, in der die Kommandozeilenparameter überprüft werden, eine Instanz des Grammatik-Generators erzeugt und initialisiert sowie die Erzeugung der Grammatik angestoßen wird. Den Inhalt der Konfigurationsdatei nimmt eine Instanz der Klasse `EBNF2ANTLR_Config` auf, die später zur Abfrage der Parameter dient.

Eine Instanz des Grammatik-Generators stellt eine Variable und eine `HashMap` zum Zugriff auf die Regeln bereit. Die Variable verweist auf die Wurzel des Syntaxbaumes und kann somit zur sequentiellen Abarbeitung sämtlicher Regeln genutzt werden. Die `HashMap` ermöglicht, eine bestimmte Regel anhand ihres Namens aufzufinden. Beide Möglichkeiten des Zugriffs werden von vielen Methoden verwendet.

Die eigentliche Erzeugung der Grammatik leistet die Methode `generateGrammar()`. Sie führt verschiedene Verarbeitungsschritte aus, die schließlich zur Grammatik und zu den Prüfwerkzeug-Klassen führen.

3.3.1 Verarbeitungsschritte

Die folgenden Abschnitte beschreiben die Verarbeitungsschritte zur Übersetzung einer EBNF-Grammatik in eine ANTLR-spezifische Grammatik. Diese sind durch die Konfigurationsdatei steuerbar. Die Verbindung von den Parametern zu den einzelnen Schritten wird jeweils hergestellt.

3.3.1.1 Parsen der Ausgangs-Grammatik

Der Dateiname der zu übersetzenden Grammatik muss hinter dem Eintrag `input_file` spezifiziert werden. Zum Parsen der DMIS-Grammatik im EBNF-Format wird ein EBNF-Parser verwendet, der mit ANTLR aus einer EBNF-Grammatik generiert worden ist. Diese wurde aus dem entsprechendem ISO-Standard entwickelt und anschließend an das davon abweichende EBNF der DMIS-Grammatik angepasst. Im Laufe der Entwicklung des Grammatik-Generators wurden weitere Änderungen vorgenommen, um ANTLR-spezifische Schreibweisen in einer EBNF-Grammatik verwenden zu können. Dies ist nötig, um bei einzelnen Regeln ANTLR steuern zu können. Die

Entwicklung des EBNF-Parsers soll nicht weiter betrachtet werden, weil das zu weit vom Thema der Arbeit wegführt.

Während des Parsens wird eine Baumdarstellung der DMIS-Grammatik erstellt, auf der alle übrigen Verarbeitungsschritte aufbauen. Dazu werden die im Entwurf erläuterten Klassen als Knoten verwendet.

3.3.1.2 Aufnehmen eigener Regeln

Um die Grammatik an die Erfordernisse des gewählten Parser-Generators anpassen zu können, kann eine Datei mit Regeln zusätzlich zur eigentlichen Grammatik angegeben werden (Option `ADDITIONAL_RULES`). Diese Regeln müssen ebenfalls im EBNF-Format vorliegen. Dadurch kann der EBNF-Parser benutzt werden, um diese zusätzlichen Regeln in die erforderliche Baumdarstellung zu bringen.

Ist eine Regel mit dem Namen einer neuen Regel bereits vorhanden, wird sie durch die neue ersetzt. Das hat den Vorteil, dass Parser-Generator-spezifische Anpassungen einzelner Regeln nicht in der Ausgangs-Grammatik vorgenommen werden müssen. Diese bleibt also unabhängig von einem bestimmten Werkzeug.

3.3.1.3 Ersetzen von Regelreferenzen

Diese Funktion erlaubt das Ersetzen aller Referenzen auf eine Regel durch Referenzen auf eine andere Regel. Dies ist zum Beispiel nötig, wenn zwei Regeln des Lexers, die in Regeln des Parsers referenziert werden, den gleichen Aufbau besitzen. Zwischen diesen beiden Regeln könnte der Lexer nicht entscheiden und würde daher eine Mehrdeutigkeitswarnung ausgeben.

Die Namen entsprechend zu behandelnder Regeln müssen im Abschnitt `RULES_TO_REPLACE` platziert werden, jeweils gefolgt von einem Komma und dem Namen der Regel, durch die diese ersetzt werden soll.

3.3.1.4 Löschen von Regeln

Werden gleich mehrere in Beziehung stehende Regeln durch neue Regeln ersetzt, kann es vorkommen, dass einzelne Regeln nicht mehr benötigt werden. Diese können durch das Programm gelöscht werden. Zu löschende Regeln können im Abschnitt `RULES_TO_DELETE` der Konfigurationsdatei angegeben werden.

3.3.1.5 Zuordnung der Regeln zu Scanner oder Parser

Die Aufteilung der Regeln auf Scanner und Parser stellt eine Schlüsselstelle bei der Grammatik-Generierung dar. Einerseits müssen die Abhängigkeiten zwischen den Regeln beachtet werden, das heißt, eine Scanner-Regel darf keine Parser-Regel referenzieren, und andererseits kann schon die Verschiebung einer einzigen Regel zahlreiche Probleme schaffen oder lösen. Entsprechend konnte eine vollkommen automatische Zuordnung nicht umgesetzt werden.

Vor ihrer Zuordnung wird eine Regel daraufhin untersucht, ob sie Zeichenketten mit einer Länge größer einem Zeichen, Referenzen auf sich selbst, virtuelle Token, Parser- oder Scanner-Regeln enthält. Jedes Vorkommen wird gezählt. Das Ergebnis des Zählens wird für die Entscheidung herangezogen. Regeln, die Referenzen auf sich selbst (Rekursionen), auf Parser-Regeln oder virtuelle Token enthalten, werden in jedem Fall zu einer Parser-Regel. Das gilt ebenfalls für Regeln, die mehr als drei Scanner-Regeln referenzieren oder mehr als eine Zeichenkette mit Länge größer einem Zeichen enthalten. Diese beiden Kriterien wurden durch mehrere Versuche ermittelt. Sie helfen dabei, dichter an die optimale Aufteilung zu gelangen.

Da zu Beginn des Algorithmus nicht bekannt ist, ob eine referenzierte Regel zum Scanner oder Parser gehört, müssen mehrere Läufe durchgeführt werden. Dabei können anfangs nur wenige Regeln zugeordnet werden, nämlich nur diejenigen Regeln, die keine Referenzen enthalten. Dies sind die Scanner-Regeln, die nur einzelne Zeichen enthalten, und die Parser-Regeln, die nur Zeichenketten enthalten. Als nächstes können solche Regeln zugeordnet werden, die neben Terminalen nur Referenzen auf einfachere, zuvor bereits

zugeordnete Regeln enthalten. So werden nach und nach alle Regeln verarbeitet.

Zur Zuordnung wird auch die Anzahl der bereits durchgeführten Läufe herangezogen. Regeln, die nach dem siebten Lauf nicht zugeordnet sind, können nur noch zu Parser-Regeln werden. Auch diese Festlegung entstand durch Ausprobieren. Sie bewirkt, dass die Schachtelung von Scanner-Regel-Referenzen nicht tiefer als sieben werden kann.

Ein Problem bei der Zuordnung stellen Rekursionen dar, die über mehrere Regeln gehen. Ein Beispiel dafür sind die Regeln zur Abbildung von Ausdrücken. Ein Ausdruck setzt sich aus verschachtelten Regel-Referenzen zusammen, an deren Ende wiederum eine Referenz auf einen Ausdruck stehen kann. In diesem Fall liegt ein Deadlock vor, weil zwei oder mehrere Regeln jeweils darauf warten, dass die andere zugeordnet wird. Um solche Probleme zu lösen, wurde die Schleifensteuerung erweitert. Hier wird ausgewertet, wie viele Regeln im letzten Durchlauf zugeordnet werden konnten. War es nicht möglich, wenigstens eine Regel zu zuordnen, werden die Regeln auf solche weitreichenden Rekursionen hin untersucht.

Die automatische Zuordnung kann übersteuert werden, indem in der Parameter-Datei aufgelistet wird, welche Regeln zum Scanner oder Parser zu zählen sind. Diese werden vor dem Versuch der automatischen Zuordnung wunschgemäß zugeordnet.

Kann eine Regel nicht zugeordnet werden, weil zum Beispiel eine referenzierte Regel fehlt, wird eine Meldung ausgegeben. Mit der Verarbeitung wird jedoch fortgefahren.

3.3.1.6 Ersetzung von Referenzen durch den Inhalt der referenzierten Regel

Soll eine Regel aus der Grammatik entfernt werden, kann dies vom Programm erledigt werden (Option `RULES_TO_INLINE`). Es kopiert dann den Inhalt der Regel an jede Stelle in der gesamten Grammatik, an der diese Regel referenziert wird. Die Regel `feature_label` muss zum Beispiel so behandelt werden, weil sonst der Baum, der beim Parsen erstellt wird, nicht die gewünschte Form hat, da der Knoten eines Tokens, das in einer referenzierten Regel erkannt worden ist, nicht den Knoten übergeordnet sein kann, die auf

die Referenz folgen. Bei den DMIS-Anweisungen, bei denen einem Label etwas zugewiesen wird, soll jedoch das Label die Wurzel eines Unterbaumes sein. Die Kinder-Knoten dieser Wurzel stellen den Namen des Befehls und die Parameter dar.

3.3.1.7 Prüfung auf Mehrdeutigkeiten

Der Syntaxbaum ist immer so aufgebaut, dass Regeln, Gruppen, Wiederholungen und optionale Gruppen, also alle Knoten vom Typ `RuleItemList` außer denen vom Typ `Alternative`, als Kinder nur Knoten des Typs `Alternative` haben können. Alternativen können wiederum Knoten jeden Typs außer Alternativen als Kinder haben. Dies wird in diesem Algorithmus vorausgesetzt. Man kann also zum Beispiel eine Regel als Liste von Alternativen ansehen, die mindestens eine Alternative enthält. Gibt es mehrere Alternativen, können diese teilweise einen gleichen Aufbau haben. Ist dies vom ersten Kind-Element an der Fall, kann sich daraus ein Problem für den Parser ergeben, da dieser – je nach Lookahead – nicht entscheiden kann, welche Alternative er wählen soll.

Um solche Probleme zu verhindern, kann der Syntaxbaum auf Mehrdeutigkeiten untersucht werden. Dabei werden alle Alternativen, die gleich beginnen, umformuliert – unabhängig davon, ob sich daraus ein Problem für den Parser ergäbe. Dies übernimmt die Methode `eliminateAmbigs()`, indem sie für jede im Baum enthaltene Regel die Methode `eliminateAmbig_subTree()` aufruft. Außerdem protokolliert sie, welche Regeln geändert worden sind, und fügt an geänderte Regeln einen Kommentar an, der den ursprünglichen Aufbau der Regel enthält. Bei der Umformulierung darf sich die Bedeutung nicht ändern. Es muss also eine eindeutige Baumstruktur gefunden werden, die die gleiche Bedeutung besitzt.

Die Methode `eliminateAmbig_subTree()` hat den typischen Aufbau zum rekursiven Durchwandern eines Teilbaumes. Für einfache Elemente wird nichts gemacht, da es in diesem Algorithmus um die Struktur des Baums geht. Einfache Elemente füllen nur die Struktur, beeinflussen diese jedoch nicht; sie dienen nur als Bindeglied zwischen den Listen. Stattdessen wird auf Mehrdeutigkeiten geprüft, wenn das aktuelle Element eine Alternative ist. In

diesem Fall werden die Geschwister des Knotens, welche ebenfalls nur vom Typ `Alternative` sein können, direkt mit diesem Knoten verarbeitet. Daher wird über den Merker `firstAlt` gesteuert, ob es sich um die erste Alternative in einer Liste handelt. Die Mehrdeutigkeitsprüfung wird nur bei der ersten Alternative ausgeführt, ein rekursiver Aufruf der Methode muss jedoch auch für die Kinder der übrigen Alternativen erfolgen.

Der Algorithmus verwendet drei Arrays. Das Array `cntMinEqualItems[]` enthält unter dem Index *i* die Anzahl der Elemente, die die Alternative *i* mit mindestens einer anderen Alternative gemeinsam hat. Das Array `equalItems[][]` speichert für jede Alternative *i*, ob sie Elemente mit der Alternative *k* gemeinsam hat. Den Zugriff auf eine bestimmte Alternative erleichtert das Array `alternatives[]`, da es das Durchwandern des Baums erspart.

Mehrdeutigkeiten können zwischen mindestens zwei und höchstens allen Alternativen einer Liste bestehen. Dabei müssen die betroffenen Alternativen nicht direkt aufeinander folgen. Die Methode fasst deren Elemente in der ersten Alternative, die in der Liste erscheint, zusammen, das heißt, die erste bleibt umgeformt erhalten, während die folgenden Alternativen aus dem Baum entfernt werden.

Zur Erkennung von Mehrdeutigkeiten werden die Alternativen mit allen jeweils folgenden Alternativen verglichen. Hier finden die Methoden `equalsSubTree()` der Klassen `RuleItem` und `RuleItemList` Anwendung, die rekursiv den übergebenen Teilbaum vergleichen. Anschließend enthält das Array `cntMinEqualItems[]` unter den entsprechenden Indizes die Anzahl der Elemente, die eine Alternative mit den ihr folgenden von links gezählt mindestens gemeinsam hat. Eine Anzahl von Null wird bei der Ermittlung des Minimums ignoriert und nur eingetragen, wenn keinerlei Gemeinsamkeiten bestehen.

In der folgenden Schleife wird zunächst der Index derjenigen Alternative ermittelt, die die größte Anzahl (größer Null) gemeinsamer Elemente mit den ihr folgenden Alternativen aufweist. Durch Zusammenfassen in dieser Alternative wird die entsprechende Mehrdeutigkeit behoben, die Anzahl

gemeinsamer Elemente auf Null gesetzt und der Schleifenkörper erneut betreten. Enthält `cntMinEqualItems[]` ausschließlich Nullen, ist diese Liste frei von Mehrdeutigkeiten, und die Schleife wird verlassen.

Sind im Teilbaum mehrere Mehrdeutigkeiten vorhanden, wird diejenige zuerst behandelt, bei der die Alternativen die größte Anzahl gemeinsamer Elemente aufweisen. Dadurch werden Mehrdeutigkeiten so früh wie möglich eliminiert. Das folgende Beispiel soll dies verdeutlichen:

Alternative 1: A X
Alternative 2: A Y
Alternative 3: A B C D E
Alternative 4: A B C F G

Alle vier Alternativen haben das erste Element gemeinsam. Die Alternativen 3 und 4 haben noch zwei weitere, also insgesamt drei Elemente gemeinsam. Nach geeigneter Umformulierung ist Alternative 4 in die dritte integriert:

Alternative 1: A X
Alternative 2: A Y
Alternative 3: A B C (D E | F G)

Nach einer weiteren Umformung sind alle Mehrdeutigkeiten entfernt:

Alternative 1: A (X | Y | B C (D E | F G))

Wäre die zweite Umformung zuerst durchgeführt worden, hätte sich folgendes ergeben:

Alternative 1: A (X | Y | B C D E | B C F G)

Die zweite Mehrdeutigkeit wäre auf eine tiefere Ebene im Baum gerückt worden, also an dieser Stelle für den Algorithmus nicht mehr greifbar. Sie würde zwar auf jeden Fall behandelt, aber das erste Verfahren hat den Vorteil, dass aus dem Aufwand des Vergleichens der größte Nutzen gezogen wird. Je mehr gleiche Elemente auftreten, um so größer ist der Vorteil. Denn bei einem erneuten Vergleich der Alternativen würde beim ersten Element festgestellt,

dass keine Mehrdeutigkeit vorliegt, anstatt dass für mehrere Elemente ein Vergleich des Teilbaums angestoßen würde.

Um den Baum ändern zu können, ohne seine Bedeutung zu ändern, muss zuvor ermittelt werden, ob mindestens eine der Alternativen nur aus den mit den anderen gemeinsamen Elementen besteht. Ist dies der Fall, muss den gemeinsamen Elementen eine optionale Gruppe folgen. Ein Beispiel hierfür ist die Regel `math_sym` des DMI-Formates:

```
<math_sym> ::= '+' | '-' | '*' | '/' | '*' '*'
```

Die dritte und fünfte Alternative beginnen jeweils mit einem Sternchen. Die dritte besteht sogar nur aus diesem einen Sternchen. Die Methode `eliminateAmbig_subTree()` liefert folgendes Ergebnis:

```
<math_sym> ::= '+' | '-' | '*' ['*'] | '/'
```

Den vorher gewonnenen Erkenntnissen entsprechend wird eine Gruppe erzeugt und mit den Elementen der Alternativen gefüllt, die auf die gemeinsamen Elemente folgen. Dabei werden nur Alternativen berücksichtigt, die weitere Elemente besitzen.

Diese Gruppe wird anschließend an die Elemente der ersten Alternative angehängt, die allen Alternativen gemeinsam sind. Die dort eventuell zusätzlichen Elemente werden dabei entfernt, da sie in der Gruppe enthalten sind. Die Alternative zu nehmen, die beim Durchwandern des Teilbaums zuerst angetroffen wird, bzw. im Array `alternatives[]` unter dem kleinsten Index abgelegt ist, hat den Vorteil, dass keine Sonderbehandlung zum Entfernen der allerersten Alternative aus dem Baum erforderlich ist. Um das erste Kind eines Knotens zu entfernen, muss nämlich dieser Knoten angesprochen werden, während bei allen folgenden Kindern der jeweilige Vorgängerknoten angefasst werden muss.

Zum Abschluss werden alle in die Umformung eingeflossenen Alternativen außer der ersten aus dem Teilbaum entfernt, und die Arrays aktualisiert.

Die Prüfung auf Mehrdeutigkeiten kann über die Option `CHECK_AMBIG` an- und abgeschaltet werden.

3.3.1.8 Festlegung zu schützender Lexer-Regeln

Wiederholt sich ein Regelteil in mehreren Regeln, ist es eventuell sinnvoll, aus diesem eine eigene Regel zu erstellen. Steht der gleiche Teil jedoch zuvor auch nur in einer Regel am Anfang, würde dadurch eine Mehrdeutigkeit eingebaut.

Bei ANTLR gibt es die Möglichkeit, Regeln als `protected` zu deklarieren. Solche Regeln führen nicht zur Erstellung eines Tokens für den Parser. Stattdessen sucht der Lexer nach einer anderen Regel, die diese Regel benutzt, nicht geschützt ist und somit zu einem Token führen kann.

Die Methode `examineLexerRuleReferences()` deklariert alle Regeln als `protected`, die nur von Lexer-Regeln referenziert werden. Sie wird aufgerufen, wenn die Option `PROTECT_LEXER_RULES` eingeschaltet ist.

3.3.1.9 Ersetzen von einzelnen Zeichen in Parser-Regeln

ANTLR lässt in Parser-Regeln keine Terminale zu, die nur aus einem Zeichen bestehen. In DMIS kommen jedoch einzelne Zeichen in den Anweisungen als Trennzeichen vor. Daher generiert die Methode `genLexerRulesForTerminals()` je eine Scanner-Regel für alle verschiedenen einzelnen Zeichen, die in den Parser-Regeln vorkommen.

3.3.1.10 Prüfung auf reservierte Wörter

Bei der Implementierung eines rekursiv absteigenden Parsers, wie zum Beispiel von ANTLR erzeugt, wird für jede Regel eine Methode erstellt. ANTLR nimmt als Namen für diese Methode den Namen der Regel. Ist dies ein reserviertes Wort der Programmiersprache, für die der Quellcode generiert werden soll, gibt es Probleme beim Compilieren. Daher werden alle Regeln in der Methode `checkForReservedWords()` geprüft. Dazu werden die reservierten Wörter der Zielsprache in einem `HashSet` abgelegt, mit deren Hilfe Regelnamen und Regelreferenzen überprüft werden. Wurde ein solches Wort verwendet, wird es durch die Methode `maskReservedWord()` geändert.

3.3.1.11 Hinweise für den Benutzer

Der Grammatik-Generator soll nicht nur Änderungen durchführen, sondern auch bei der Suche nach möglichen Problemen in der Grammatik helfen. Dazu

wird der Baum untersucht. Dies geschieht jedoch nur, wenn alle Regeln dem Scanner oder dem Parser zugeordnet worden sind, da andernfalls Inkonsistenzen im Baum vorliegen. Unter diesen Umständen machen die momentan implementierten Tests keinen Sinn.

3.3.1.11.1 Nicht referenzierte Regeln

Auf den ersten Blick scheint es nahe liegend, nicht referenzierte Regeln zu löschen, zumal dies nicht in der Original-Grammatik, sondern in der neu erstellten geschehen würde. Es gibt jedoch Regeln, die von keiner anderen Regel referenziert werden, und trotzdem eine Daseinsberechtigung haben. Daher wird eine solche Regel nicht automatisch gelöscht, sondern es wird ein Hinweis für den Benutzer ausgegeben, dass die Regel von keiner anderen Regel verwendet wird.

Ein Beispiel hierfür ist die Startregel einer Grammatik wie zum Beispiel die Regel `output_file` beim DMO-Format. Da in der Konfigurationsdatei angegeben werden muss, welche Regel die Startregel ist, und dies somit bekannt ist, wird der Hinweis an dieser Stelle allerdings unterdrückt.

Beim DMO-Format muss jedoch die Regel `word` für den Lexer ergänzt werden, auch wenn sie nie referenziert wird. Der Grund hierfür wird im Abschnitt zur Anwendung des Grammatik-Generators erläutert.

3.3.1.11.2 Regeln mit gleichem Inhalt

Haben zwei Regeln den gleichen Aufbau, kann das zu Problemen führen. Wenn es sich bei diesen Regeln zum Beispiel um Lexer-Regeln handelt, kann dieser nicht entscheiden, welche der beiden zu nehmen ist, also welcher Tokentyp an den Parser zu melden ist. Auch bei Parser-Regeln können Probleme durch inhaltlich gleiche Regeln auftreten. Werden diese an einer Stelle als zwei Alternativen verwendet, kann der Parser keine Entscheidung treffen. Im Parser müssen aber solche Regeln nicht zu Problemen führen. Ob eine Änderung nötig ist, kann das Programm bisher jedoch nicht feststellen. Aber auch wenn es dies könnte, wäre eine Änderung schwierig. Daher wird die Entscheidung, was zu tun ist, dem Benutzer überlassen.

3.3.1.12 Schreiben der ANTLR-Grammatik

Das Ergebnis der Manipulationen soll eine von ANTLR verwendbare Grammatik sein. Die Erstellung einer solchen Datei ist der Klasse `ANTLRGrammarWriter` überlassen.

3.3.1.13 Generierung von Prüfwerkzeug-Klassen

Es werden Klassen für die Verwendung mit dem DMIS-Checker generiert, und zwar die Adapter-Klasse für den Parser, ein Präprozessor und ein Semantik-Prüfer. Diese Dateien müssen in die jeweiligen Unterverzeichnisse des DMIS-Checker-Verzeichnisses kopiert werden. Nach dem Compilieren dieser neuen Werkzeuge kann sofort eine Datei auf die entsprechende DMIS-Version geprüft werden. Das ist möglich, weil für alle drei Werkzeugarten eine Klasse für die entsprechende DMIS-Version zur Verfügung steht. Der Präprozessor übernimmt die Standard-Implementierung seiner Oberklasse. Der Semantik-Prüfer lässt sich ebenfalls starten, prüft jedoch nichts. Dieser muss manuell um Prüfungen – eventuell von Klassen anderer DMIS-Versionen – erweitert werden.

Zur Bildung der jeweiligen Klassennamen müssen in der Parameterdatei die Version (`MAJOR_VERSION`, `MINOR_VERSION`) und das Format (`TYPE`) spezifiziert werden.

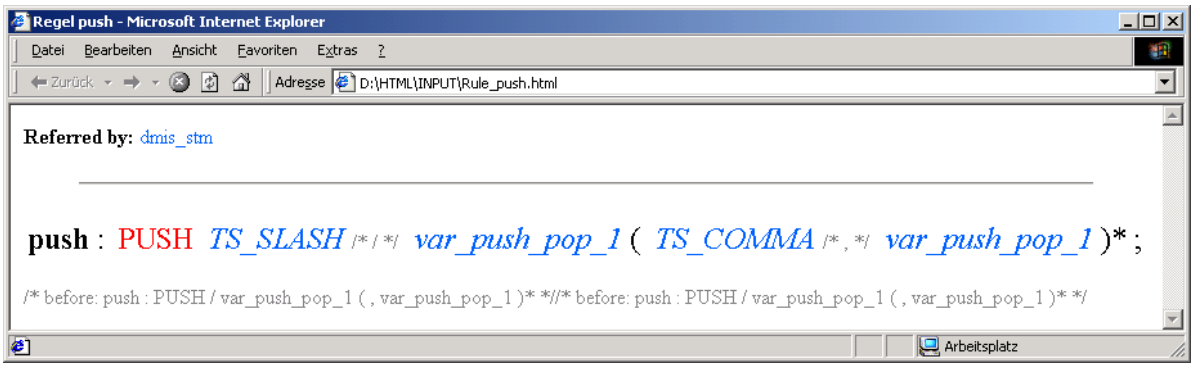
3.3.1.14 Schreiben einer modifizierten EBNF-Grammatik

Um die modifizierte mit der ursprünglichen Grammatik vergleichen und gegebenenfalls Verbesserungsvorschläge an das Standardisierungsgremium richten zu können, wird zusätzlich eine Grammatik im EBNF-Format erzeugt. Hier kommt die Klasse `EBNFGrammarWriter` zum Einsatz.

3.3.1.15 Generierung der Grammatik-Dokumentation in HTML

Der Umfang der Grammatiken erschwert die Suche nach bestimmten Regeln und das Verfolgen von Beziehungen zwischen den Regeln. Daher stellt sich das Verständnis einer gesamten Grammatik umso schlechter ein. Um dem entgegen zu wirken, kann man sich eine Dokumentation zu einer generierten ANTLR-Grammatik erstellen lassen. Diese umfasst eine Liste aller Regeln, untergliedert nach Parser- und Lexer-Regeln, und je Regel eine HTML-Seite

mit dem jeweiligen Aufbau, der durch unterschiedliche Schriftstile verdeutlicht wird. Im Kopf jeder Seite sind alle Regeln aufgelistet, die die dargestellte Regel referenzieren. Die Regeln dieser Liste und alle referenzierten Regeln sind verlinkt, so dass die Beziehungen zwischen den Regeln leicht erkundet werden können. Auf der Übersichtsseite wird die Startregel der Grammatik vor den anderen Regeln zusätzlich genannt. Ferner sind geschützte Lexer-Regeln als solche gekennzeichnet.



Quelle: Eigene Darstellung

Darstellung 18: HTML-Dokumentation

Die Dokumentation der Anweisung `PUSH` ist beispielsweise in Darstellung 18 zu sehen. Wie oben beschrieben, würde ein Klick auf `var_push_pop_1` genügen, um den Aufbau der Regel und damit die möglichen Parameter der `PUSH`-Anweisung anzusehen.

Die Erzeugung der Dokumentation kann über die Option `GEN_HTML_DOC` in der Parameterdatei verhindert werden. Andernfalls muss dort ein Verzeichnis angegeben werden, in dem sämtliche HTML-Dateien abgelegt werden (Option `HTML_DOC_DIR`).

3.3.1.16 Anwendung von ANTLR auf die generierte Grammatik

Zur Vereinfachung für den Benutzer lässt der Grammatik-Generator die zuvor von ihm erzeugte Grammatik sofort von ANTLR verarbeiten. Die Anzahl der von ANTLR aufgeführten Warnungen wird gegebenenfalls ausgegeben. Sollten Fehler oder Warnungen aufgetreten sein, können diese in der Datei „antlr_output.txt“ angesehen werden. Der automatische Start von ANTLR

kann in der Konfigurationsdatei über die Option `RUN_ANTLR` abgeschaltet werden.

3.3.2 Klassen zur Erzeugung der Grammatiken

Die Erzeugung eines bestimmten Grammatik-Formats übernimmt jeweils eine Klasse. Die allen gemeinsame abstrakte Oberklasse `GrammarWriter` stellt Methoden zur Formatierung der Datei zur Verfügung. Sämtliche dieser Klassen sind in dem Package `grammarWriters` enthalten.

3.3.2.1 ANTLR-Grammatik

Die Klasse `ANTLRGrammarWriter` erstellt aus dem Baum mit den Regeln eine ANTLR-Grammatik. Dabei müssen verschiedene ANTLR-Spezifika beachtet werden. Ferner wird Java-Quelltext ergänzt, der aus der Konfigurationsdatei entnommen oder generiert wird.

Zunächst werden das Package und die Imports in die Datei geschrieben. Diese gelten für die Parser- und die Scanner-Klasse. Darauf folgt der Block mit den Informationen zur Generierung des Parsers, der durch eine Zeile mit dem Namen der zu generierenden Parser-Klasse eingeleitet wird. Die Klassennamen für Parser und Scanner setzt die Klasse `DMISCheckerToolFactory` aus Einträgen der Konfigurationsdatei zusammen, nämlich aus `TYPE`, `MAJOR_VERSION` und `MINOR_VERSION`. Daran schließen sich ein Block mit Optionen, ein Block mit Java-Quellcode und die Regeln für den Parser an. Die Optionen werden aus der Konfigurationsdatei entnommen (`PARSER_LOOKAHEAD`, `BUILD_AST`). Außerdem wird die Generierung von Quellcode zur Fehlerbehandlung durch ANTLR abgeschaltet, damit das Wiederaufsetzen nach einem Fehler möglich wird. Der Quellcode-Block umfasst genau den Inhalt der Datei, deren Name unter der Option `ADDITIONAL_PARSER_CODE` angegeben wurde. Sie kann zum Beispiel Variablen-Deklarationen oder Methoden enthalten. Dieser Quellcode wird von ANTLR als erstes in den Rumpf der Parser-Klasse eingefügt.

Einzelne Regeln können ebenfalls einen Block mit Optionen oder Java-Quellcode enthalten. Entweder waren diese Angaben in der Ausgangsgrammatik enthalten oder sie werden aus Einträgen der

Konfigurationsdatei generiert. Aus dem Eintrag PARAPHRASE, der für jede einzelne Regel spezifiziert werden kann, wird die entsprechende ANTLR-Option generiert. Enthält eine Regel eine Wiederholung mit einer Obergrenze, wird Quellcode zur Prüfung dieser Grenze generiert. Dies kann mit der Option GEN_LENGTH_CHECK abgeschaltet werden. Um Probleme beim Compilieren zu verhindern, werden dabei Variablen mit in der Klasse eindeutigen Namen verwendet. Die Eindeutigkeit wird durch einfaches Durchnummerieren im Namen der Variablen erreicht.

Mit der Option EXCEPTION_HANDLING_CODE kann der Name einer Datei angegeben werden, die Quellcode zur Exception-Behandlung bei einzelnen Regeln enthält. Dieser Quellcode wird im Anschluss an die entsprechenden Regeln ausgegeben. Ferner werden gemäß den Angaben in den Abschnitten TREE_ROOTS und TREE_IGNORE der Konfigurationsdatei die erforderlichen Markierungen für die Baumerstellung vorgenommen.

Nach dem Block mit den Angaben zum Parser folgt der Block mit den Angaben zum Scanner. Dieser hat im Prinzip den gleichen Aufbau wie der Block für den Parser.

3.3.2.2 EBNF-Grammatik

Die EBNF-Grammatik erzeugt die Klasse EBNFGrammarWriter. Momentan werden lediglich sämtliche Regeln der Grammatik in der Reihenfolge der ursprünglichen Grammatik im EBNF-Format ausgegeben.

3.3.3 Anwendung

In diesem Abschnitt soll gezeigt werden, wie mit Hilfe des Grammatik-Generators aus der in der Standard-Dokumentation enthaltenen Grammatik jeweils ein Parser für das Eingabe- und Ausgabe-Format von DMIS 04.0 erzeugt wurde.

Dabei sind verschiedene Probleme aufgetreten. So waren zum Beispiel einige syntaktische Fehler in der Grammatik enthalten. Am meisten Probleme haben jedoch Mehrdeutigkeiten bereitet. Außerdem konnten nicht alle in der Grammatik enthaltenen semantischen Informationen in die ANTLR-spezifische Grammatik übernommen werden.

Eine mögliche Ausgabe des Grammatik-Generators ist in Anhang I dargestellt. Es handelt sich dabei um die Meldungen, die bei der Generierung einer noch nicht optimalen Grammatik ausgegeben wurden, um ein möglichst anschauliches Beispiel darzustellen. Insbesondere steht hier am Ende der Ausgabe die Anzahl der von ANTLR ausgegebenen Warnungen, damit nach dem Durchlaufen des Grammatik-Generators sofort erkannt werden kann, ob noch Mehrdeutigkeiten vorliegen. Die für DMIS Version 04.0 erstellten Parser basieren auf Grammatiken, bei denen dies nicht der Fall ist. Sie sind in diesem Sinne optimal.

Für jeden Verarbeitungsschritt sind statistische Informationen zu finden. So wurden zum Beispiel die 2.989 Zeilen[1] der DMIS-Input-Grammatik innerhalb von knapp einer Sekunde geparst. Des Weiteren kann abgelesen werden, wie viele Regeln in der Grammatik enthalten sind, ersetzt, hinzugefügt oder gelöscht wurden. Die Zuordnung der Regeln zu Scanner und Parser konnte in 11 Schleifendurchläufen mit dem Ergebnis abgeschlossen werden, dass es 34 Scanner- und 424 Parser-Regeln geben wird. Weiter unten sind Hinweise für den Benutzer zu sehen („notice“).

		DMI		DMO	
		Anzahl Regeln	%	Anzahl Regeln	%
ANTLR-Grammatik	• original	342	68,1	202	67,8
	• ersetzt	74	14,7	32	10,7
	• ergänzt	26	5,2	7	2,3
	• generiert	60	12,0	57	19,1
	• gelöscht	44		21	
	gesamt	502	100,0	298	100,0
EBNF-Grammatik		460	100,0	255	100,0

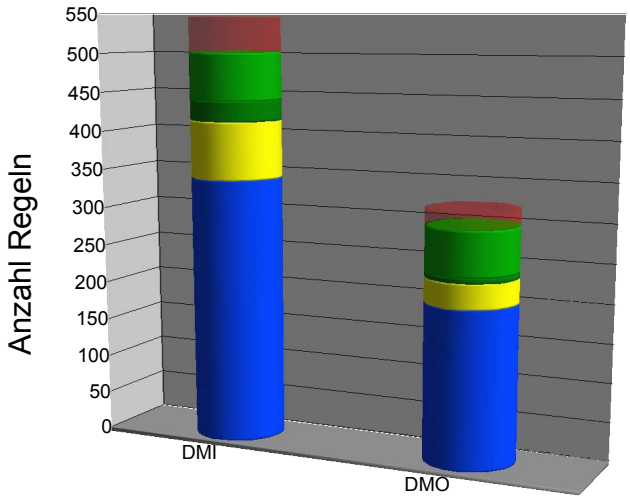
Quelle: Eigene Darstellung

Darstellung 19: Anzahl der Regeln in den Grammatiken

Die Tabelle in Darstellung 19 zeigt die Herkunft der Regeln, aus denen sich die ANTLR-Grammatiken für DMIS 04.0 zusammensetzen. Zum Vergleich ist die Anzahl der Regeln der EBNF-Grammatiken aus dem Standard angegeben.

1 Die besagte Grammatik umfasst mittlerweile ein paar wenige Zeilen mehr, da bei Fehlerkorrekturen entsprechende Kommentare eingefügt wurden.

Das zahlenmäßige Verhältnis der Regeln veranschaulicht grafisch Darstellung 20. Die Farben der Säulenabschnitte werden in der Tabelle erläutert.



Quelle: Eigene Darstellung

Darstellung 20: Verhältnis der Regeln der ANTLR-Grammatiken

Es wurden bei beiden Grammatiken fast 70 Prozent der Regeln aus der Ausgangsgrammatik übernommen und jeweils ca. 60 Regeln generiert. Entscheidend ist jedoch, dass beim Output-Format mehr als zehn Prozent und beim Input-Format sogar fast zwanzig Prozent der Regeln hinzugefügt oder manuell geändert worden sind, um eine erfolgreiche Parser-Generierung durch ANTLR und Fehler-Recovery zu ermöglichen.

Anhang II enthält das Protokoll der Änderungen an der Grammatik des DMIS-Input-Formats, die zur Freiheit von Mehrdeutigkeiten führten. Daraus ist zu erkennen, dass eine gewisse Erfahrung nötig ist, um eine brauchbare Grammatik zu erhalten. Hervorzuheben ist zum Beispiel, dass das Einschalten der Grammatik-Generator-Option `PROTECT_LEXER_RULES` die Anzahl der Mehrdeutigkeiten im Scanner von 2.136 auf 1.124 beinahe halbiert hat (lfd. Nr. 2). Außerdem wird deutlich, dass die Erhöhung des Lookaheads sowohl für den Parser (pk) als auch für den Scanner (lk) von sieben auf zehn keine

Verringerung der Anzahl aufgetretener Warnungen bewirkt hat (lfd. Nr. 8 und 9). Die Verbesserung des Algorithmus zur Auffindung von Mehrdeutigkeiten scheint nur einen sehr geringen Nutzen gebracht zu haben (lfd. Nr. 67). Der Grund dafür ist, dass die meisten durch den Algorithmus entfernten Mehrdeutigkeiten bereits durch die Erhöhung des Lookaheads behandelt worden sind. Auf Grund des verbesserten Algorithmus konnte jedoch der Lookahead verringert werden, was allerdings nicht aus dem Protokoll hervor geht, da die letzten Schritte nicht mehr im Detail aufgeführt sind.

3.3.3.1 DMIS 04.0 Output-Format

Die Grammatik für das Output-Format wurde zuerst erzeugt, da sie einen geringeren Umfang und eine geringere Komplexität besitzt als die des Input-Formats.

3.3.3.1.1 Zusätzliche Regeln

Aus folgenden Gründen wurden eigene Regeln definiert:

1. Zur Anpassung an ANTLR-spezifische Gegebenheiten
2. Zur Vermeidung von Mehrdeutigkeitswarnungen
3. Zur Unterscheidung von Zahlen-Typen und Wertebereichen
4. Zur Ermöglichung von Fehler-Recovery

Insgesamt wurden 39 Regeln definiert, von denen 32 Regeln eine Regel der Ausgangsgrammatik ersetzen. Dieser Abschnitt erläutert nur einige dieser Regeln, die jedoch als Beispiel für die wichtigsten Änderungen stehen.

Die Regel `EOL` wurde definiert, um bei jedem Zeilenumbruch die interne Zeilennummer des Scanners inkrementieren zu können. Außerdem ist die Schreibweise der Zeichen eines Zeilenumbruchs Parser-Generator-spezifisch und es gab keine Regel, die diese beiden Zeichen enthalten hat. Da jedoch unzulässige Zeichen nicht ignoriert, sondern gemeldet werden sollen, muss jedes zulässige Zeichen explizit in einer Regel erscheinen.

Zur Vermeidung von Mehrdeutigkeiten wurden einige Regeln aus der Ausgangsgrammatik kopiert und mit ANTLR-Optionen wie zum Beispiel `greedy` oder syntaktischen Prädikaten versehen, um die Code-Generierung zu beeinflussen. Ebenfalls ein Mehrdeutigkeitsproblem bestand mit den

vorhandenen Regeln zur Erkennung der verschiedenen Zahlen-Typen und Wertebereiche. Diese wurden komplett ersetzt. Der Scanner besitzt nur noch eine Regel (`POSITIVE_INTEGER`), in der mit semantischen Aktionen abhängig von bestimmten Zeichen der Typ des Tokens verändert wird. Im Parser gibt es mehrere Regeln, die eine Hierarchie von Typen und Wertebereichen abbilden. Sie stellen eine Liste von zulässigen Token dar. Die Unterscheidung von Typen und Wertebereichen stellt bereits eine semantische Prüfung dar, die ohne die Möglichkeiten von ANTLR nicht machbar gewesen wäre. Diese Vermischung von syntaktischen und semantischen Prüfungen ist zwar weniger elegant, sie ist aber auch eine sehr einfache Möglichkeit diese Prüfungen durchzuführen, da die entsprechenden Informationen, zum Beispiel welcher Typ zulässig ist, in der Ausgangsgrammatik enthalten sind. Die Alternative wäre die manuelle Umsetzung jeder einzelnen dieser Prüfungen im Semantik-Prüfer.

Um die Analyse nach einem Fehler fortführen zu können, musste vor allem die Regel `output_file` geändert werden. Diese enthält nun semantische Aktionen, die nach einem Fehler das Fortsetzen der Prüfung bewirken, nämlich eine Schleife. Für die von der Regel `output_file` referenzierten Regeln `out_first_stmt`, `dmis_seq` und `out_last_stmt` wurde jeweils Fehlerbehandlungsquellcode angegeben, so dass die entsprechenden Methoden des Parsers eventuell auftretende Fehler behandeln und die Steuerung an die Methode `output_file()` zurückgeben. Tritt ein Fehler auf, wird er durch einen Aufruf der Methode `reportError()` protokolliert. Anschließend werden alle Token bis zum nächsten Zeilenumbruch konsumiert, und die Methode zugunsten von `output_file()` verlassen. Die Schleife sorgt dafür, dass solange erneut versucht wird, eine DMIS-Anweisung zu erkennen, bis die Anweisung `ENDFIL` erkannt wurde oder das Dateiende erreicht ist.

Mit der Regel `word` wurde eine Regel hinzugefügt, die die Rolle des klassischen Bezeichners in Programmiersprachen übernimmt, da es andernfalls keine solche Regel gäbe. Sie ist aber erforderlich, damit der Scanner ein mit einem Buchstaben beginnendes Wort erkennen kann. Das erkannte Wort vergleicht er anschließend mit seiner Symboltabelle. Wird es darin gefunden,

wird der Tokentyp auf den entsprechenden Typ gesetzt. Andernfalls wird das gefundene Wort selbst an den Parser weitergereicht. Da dies dort nirgendwo referenziert wird, gibt dieser eine Fehlermeldung aus. Ohne diese Regel müsste jedoch für jedes Terminal, also auch die mit einer Länge größer einem Zeichen, eine Regel im Scanner erstellt werden, damit diese erkannt werden könnten. Außerdem würde bei einem an unzulässiger Stelle stehenden Wort der erste Buchstabe vom Scanner nicht zugeordnet werden können, so dass er mit einer Fehlermeldung abbrechen würde.

3.3.3.1.2 Zusätzlicher Java-Quellcode

Neben einigen Variablen-Deklarationen und Methoden zur Speicherung und zum Zugriff auf statistische Informationen wurde die Methode `reportError(String)` überschrieben, um die Anzahl der ausgegebenen Fehlermeldungen zu zählen. Auch die Methode `reportError(RecognitionException)` wurde überschrieben, um die Ausgabe zu erweitern. Hier soll der Name der Methode, in der die Exception aufgetreten ist, ergänzt werden. Dazu wird die Methode `locationInfo()`[1] aufgerufen, die versucht, durch Parsen des Stack-Trace den Namen der Methode zu extrahieren, in der die Exception aufgetreten ist. Da für jede Regel der Grammatik eine Methode mit dem gleichen Namen im Parser erzeugt wird, war die ursprüngliche Idee, die DMIS-Anweisung zu ermitteln, in der ein Fehler aufgetreten ist. Dies wird durch die Tatsache begünstigt, dass die Namen von Regeln, die in einer Regel für eine DMIS-Anweisung referenziert werden, also die Namen solcher Regeln, die mögliche Parameter beschreiben, meist mit einer Ziffer enden, während die DMIS-Anweisungen (Major-Words, Minor-Words) nie mit einer Ziffer enden. Die eigentliche DMIS-Anweisung kann jedoch nicht immer ermittelt werden.

3.3.3.2 DMIS 04.0 Input-Format

Bei der Erzeugung der Grammatik für das Input-Format konnten viele Änderungen aus der Grammatik des Output-Formats übernommen werden. Die Regeln zur Zahlenerkennung mussten jedoch wieder komplett neu erstellt werden, da die Vorzeichen („+“, „-“) als Operatoren in Ausdrücken erlaubt

1 Die Methode stammt ursprünglich von dem Open-Source-Projekt log4j (http://logging.apache.org/log4j/docs/index.html).

sind und somit als Symbol dem Parser gemeldet werden müssen. Die Unterscheidung der Wertebereiche kann jedoch im erforderlichen Rahmen auch im Parser erfolgen. Welche dieser beiden Varianten die besseren Fehlermeldungen liefert, muss die Praxis zeigen.

Die Ausdrücke stellen das einzige Sprachkonstrukt dar, für das nicht alle in der Ausgangsgrammatik enthaltenen Informationen erhalten werden konnten. Das hat zur Folge, dass eine Überprüfung der in einem Ausdruck enthaltenen Typen durch den Parser nicht möglich ist. Der Grund ist die beliebig tiefe Schachtelung von geklammerten Ausdrücken, die dazu führt, dass das entscheidende Symbol zur Festlegung des Ergebnistyps eines Ausdrucks zumindest theoretisch nicht sicher nach endlich vielen Symbolen gefunden wird. Die einzige DMIS-Anweisung, bei der ebenfalls Informationen verloren gegangen sind, ist `DATSET`.

Die Funktion der Regel `word` aus der DMO-Grammatik erfüllt in der DMI-Grammatik die Regel `varname`. Hier ist also keine zusätzliche Regel dafür erforderlich.

4 Testen

Der DMIS-Checker wurde im Projektverlauf nicht explizit getestet. Nur die während vereinzelter Anwendungen aufgetretenen Fehler wurden lokalisiert und behoben. Dieses Kapitel stellt daher kein Testprotokoll dar, sondern zeigt Probleme auf, die bei diesem Projekt über die üblichen Probleme beim Testen objektorientierter Software hinaus bestehen. Als Einleitung folgt ein Bericht aus der Praxis der DMIS-Konformitätsprüfung mit dem DMIS-Checker.

Beim Kontrollieren eines Fehlerberichts stellte sich bereits die zweite Fehlermeldung als falsch heraus. Dies resultierte aus einem Fehler in der Grammatik des Standards, wo zwei Alternativen nicht voneinander getrennt waren. Nachdem der Fehler korrigiert und der Grammatik-Generator durchgelaufen war, stand in dessen Ausgabe zu lesen, dass ANTLR eine Mehrdeutigkeitswarnung ausgegeben hatte. Demnach sollte nicht zwischen zwei Regeln unterschieden werden können. Die Ursache hierfür war jedoch ein weiterer Fehler in derselben Regel, nämlich ein falsches Terminal. Beim Untersuchen der Mehrdeutigkeitswarnung wurde allerdings auch in der anderen betroffenen Regel ein Fehler gefunden. Dort waren ebenfalls zwei Alternativen nicht getrennt.

Dieser Bericht aus der praktischen Anwendung des DMIS-Checkers weist auf das größte Problem hin: Es ist nicht sicher, dass die Ausgangsgrammatik den Standard abbildet. Dies zu überprüfen stellt eine sehr aufwändige Aufgabe dar. Sind jedoch Fehler in der Grammatik, kann der DMIS-Ckecker nicht vollkommen korrekt arbeiten, das heißt, er gibt eventuell fälschlicherweise Fehlermeldungen aus oder findet bestimmte Fehler nicht. Überflüssige Fehlermeldungen wären jedoch harmlos im Vergleich zu übersehenen Fehlern.

Ein weiteres Problem ist, dass der Standard an bestimmten Stellen ungenau ist, so dass jeder Anwender gezwungen ist, seine eigene Interpretation zu Grunde zu legen. Auf einer eigenen Interpretation kann jedoch kein Prüfwerkzeug, geschweige denn ein Interpreter aufbauen.

Durch die Verwendung eines Generators kommt eine weitere mögliche Fehlerquelle hinzu. Der Generator könnte fehlerhaft sein und daher auch eine fehlerhafte Ausgabe liefern. Auch wenn funktionsfähiger Quellcode generiert

wird, der einige Tests besteht, ist nicht sichergestellt, dass alle möglichen Eingaben korrekt umgesetzt werden. Der Umfang der DMIS-Grammatiken erschwert eine Überprüfung der Korrektheit erheblich. Eine solche Prüfung müsste jedoch bei diesem Projekt sogar für die Ausgaben zweier Generatoren durchgeführt werden, nämlich für die des Grammatik-Generators und die des Parser-Generators.

Das Testen von Prüfungen, die manuell für einen Semantik-Prüfer erstellt wurden, birgt ähnliche Probleme. Zum Einen können semantische Prüfungen umfangreich, kompliziert oder beides sein, und zum Anderen ist bekanntlich die Semantik von DMIS nicht formal festgelegt. Es muss also gegen eine textuelle Beschreibung getestet werden, die regelmäßig Ungenauigkeiten enthält.

D Anwendungsbeispiel für den DMIS-Checker

Die Arbeitsweise des DMIS-Checkers soll ein Beispiel verdeutlichen. Um eine Datei mit dem Namen „Beispiel.dmi“ auf Einhaltung des DMIS-04.0-Standards zu prüfen, wird das Programm auf der Kommandozeile folgendermaßen aufgerufen:

```
java DMIS_Checker Beispiel.dmi 4 0
```

Der erste Parameter ist der Name der zu prüfenden Datei, der zweite die Hauptversionsnummer und der dritte die Nebenversionsnummer. Ob auf DMIS-Input oder DMIS-Output zu prüfen ist, wird aus der Dateiendung ermittelt, in diesem Beispiel also DMIS-Input.

Das Messprogramm „Beispiel.dmi“ ist in Anhang III abgedruckt. Die enthaltenen Fehler, die vom DMIS-Checker gefunden wurden, sind farblich markiert. Grün steht für Fehler, die durch den Präprozessor gefunden worden sind. Die vom Parser gefundenen Fehler sind rot und die vom Semantik-Prüfer gefundenen Fehler gelb markiert. Der zugehörige Fehlerbericht ist in Anhang IV zu finden.

In dem Fehlerbericht stehen zunächst der Name des Programms und der zu prüfenden Datei, damit diese dem Bericht zugeordnet werden können. Anschließend beginnt der Präprozessor seine Arbeit mit der Ausgabe seiner Versionsinformationen (Zeile 5). Darauf folgen die eigentlichen Fehlermeldungen und eine Abschlussmeldung, die die Anzahl der bearbeiteten Zeilen, die benötigte Zeit und die Anzahl der gefundenen Fehler enthält.

Wie aus den Zeilen 6 und 7 des Fehlerberichtes zu entnehmen ist, sind die Zeilen 35 und 55 jeweils um ein Zeichen zu lang. In den Zeilen 38 und 39 befindet sich eine noch längere Anweisung, die sich sogar über zwei Zeilen erstreckt. Hier ist jedoch keine Fehlermeldung ausgegeben worden, da die Anweisung mit dem Zeilenumbruchzeichen „$“ auf zwei Zeilen aufgeteilt wurde.

Die vom Präprozessor vorbereitete Datei wird als nächstes vom Parser geprüft. Er gibt ebenfalls zunächst seine Version, dann die Fehlermeldungen und eine Abschlussmeldung aus.

Die Meldung in Zeile 13 bedeutet, dass ein Komma erwartet, aber das Ende der Zeile gefunden wurde. Daraus kann man schließen, dass hier ein Parameter für den Befehl FILNAM fehlt, was ein Blick in die Dokumentation des Standards bestätigt: Es muss zusätzlich die DMIS-Version angegeben werden.[1] Ein geübter Anwender dürfte durch die Angabe, wo der Fehler aufgetreten ist, einen Fehler relativ schnell einschätzen können, auch ohne in der Dokumentation des Standards nachzuschlagen.

Zeile 15 beschreibt einen Fehler in einer TEXT-Anweisung. Dort wurde unerwarteter Weise ein Komma gefunden. Bei der Betrachtung der Zeile 17 des Messprogramms fallen zwei hintereinander stehende Kommata ins Auge. Da Kommata zur Trennung von Parametern in einer Anweisung verwendet werden, wurde hier offensichtlich der Wert eines Parameters vergessen oder ein Komma zu viel eingegeben.

Der in Zeile 16 aufgeführte Fehler betrifft ein unerwartetes Minuszeichen. Durch die Angabe „POSITIVE_REAL_VAL“ ist sofort ersichtlich, dass hier anstatt eines positiven ein negativer Wert übergeben wird. Schlägt man die Anweisung FEAT/SPHERE in der Standard-Dokumentation nach, wird auch dieser Fehler bestätigt: An der entsprechenden Stelle muss ein positiver Wert angegeben werden.[2]

Zeile 17 zeigt eine Fehlermeldung, die von Quellcode erzeugt worden ist, der durch den Grammatik-Generator generiert wurde. Variablennamen dürfen maximal 16 Zeichen lang sein.[3] Dieser zählt jedoch 17 Zeichen.

Als letztes Prüfwerkzeug verrichtet der Semantik-Prüfer seine Arbeit. Sein Ausgabeblock entspricht dem der anderen Werkzeuge, außer dass nicht die Anzahl der überprüften Zeilen, sondern der überprüften Anweisungen ausgegeben wird.

Der erste vom Semantik-Prüfer gefundene Fehler tritt in Zeile 22 auf. Dort wird der Variablen COUNT1 ein Wert zugewiesen. Laut Fehlermeldung wird diese jedoch erst in Zeile 30 deklariert, was durch Lesen des Messprogramms

1 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 209

2 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 203

3 Vgl. ANSI/CAM-I, DMIS 04.0, 2001, S. 15

nachvollzogen werden kann. Der zweite Fehler ist die Verwendung der Variablen `LV_S1X_123456789A` in der Zeile 59, die tatsächlich nirgendwo im Programm deklariert wird.

E AUSBLICK

Erste vom DMIS-Checker erstellte Fehlerberichte werden bereits in der Praxis verwendet.

Um den Nutzen des DMIS-Checkers weiter zu steigern, sollten weitere semantische Prüfungen umgesetzt werden. Ansatzpunkte hierfür kann die Beschreibung der Anweisungen in der Dokumentation des DMIS geben.

Auch von den zukünftigen Anwendern des DMIS-Checkers wurden bereits Wünsche für weitere Prüfungen genannt. Diese reichen bis hin zu logischen Prüfungen, zum Beispiel, ob eine Variable vor ihrer ersten Benutzung initialisiert wurde. Ob es sich bei einer fehlenden Initialisierung um einen Fehler handelt, wird der DMIS-Checker wahrscheinlich nie eindeutig feststellen können. Aber er kann Hinweise auf mögliche Fehlerquellen geben.

Des Weiteren bleibt abzuwarten, ob die Entwicklung des DMIS-Checkers zu einer Verbesserung der in die Standard-Dokumentation integrierten Grammatik von DMIS beitragen kann, indem mit Hilfe der vom Grammatik-Generator modifizierten EBNF-Grammatik Verbesserungsvorschläge unterbreitet werden. Vielleicht kann auch die eine oder andere Unklarheit in der Semantik beseitigt oder gar eine formale Beschreibung der Semantik entwickelt werden. Das würde möglicherweise die Generierung von semantischen Prüfungen erlauben.

Ein Einfluss auf die Grammatik würde sich auch auf die Anpassung des DMIS-Checkers an kommende Versionen von DMIS auswirken. Dies bleibt aber in jedem Fall ein Höhepunkt in der Weiterentwicklung des DMIS-Checkers.

F Fazit

Das Prüfwerkzeug DMIS-Checker kann beinahe sämtliche Sachverhalte prüfen, die in der Ausgangsgrammatik beschrieben sind, und darüber hinaus sogar einzelne Aspekte der Semantik, die nur textuell beschrieben sind. Eine Erweiterung um semantische Prüfungen ist leicht möglich dank der beim Parsen erzeugten Baumdarstellung. Diese ermöglicht außerdem die Wiederverwendung des Parsers.

Eine vollkommen automatische Generierung eines Parsers ist nicht möglich. Der entwickelte Grammatik-Generator kann jedoch weiter verbessert werden.

Das Ziel, eine Datei trotz eines Fehlers bis zum Ende zu prüfen, konnte erreicht werden. Dies birgt jedoch die Gefahr von Folgefehlern. Die Aussagekraft der Fehlermeldungen konnte durch die Angabe der Zeilen- und Spaltennummern erhöht werden. Auch der Versuch, die DMIS-Anweisung zu ermitteln, trägt einen Teil dazu bei. Hier besteht jedoch erhebliches Verbesserungspotenzial.

Die durch die Ausgangsgrammatik ermöglichten semantischen Prüfungen vereinfachen und verbessern zwar die Prüfung, haben jedoch den Nachteil, dass Anweisungen, in denen beispielsweise eine negative anstatt einer positiven Zahl steht, nicht in die Baumdarstellung aufgenommen werden. Dadurch kann die Anweisung keinen weiteren semantischen Prüfungen unterzogen werden. Es können sogar Folgefehler entstehen. Daher wäre ein besserer Ansatz, die Grammatik für die Syntaxanalyse so genau wie nötig, aber so allgemein wie möglich zu formulieren. Dies hätte eine drastische Vereinfachung der Grammatik zur Folge, da konsequenterweise auch die Prüfung der Parameter der DMIS-Anweisungen dem Semantik-Prüfer überlassen werden sollte. Dabei ginge jedoch so viel Information verloren, dass dies nur sinnvoll erscheint, wenn zusätzlich eine formale Beschreibung der Semantik erstellt würde, aus der entsprechende semantische Prüfungen generiert werden könnten. Andernfalls wäre eine leichte Anpassbarkeit nicht gegeben.

ANHANG I - AUSGABE DES GRAMMATIK-GENERATORS

```
EBNF2ANTLR
PARSING "dmis4input.grm"...
  FINISHED (2989 line(s), 464 rule(s) in 00:00,992 minutes)
PARSING AND TREATING ADDITIONAL RULES...
  FINISHED (743 line(s), 85 rule(s) replaced and 20 rule(s) added in 00:00,180
minutes)
REPLACING REFERENCES OF RULES...
  FINISHED (1 rule(s) eliminated in 00:00,000 minutes)
DELETING RULES...
  FINISHED (25 rule(s) eliminated in 00:00,010 minutes)
ASSIGNING RULES TO LEXER OR PARSER...
  FINISHED (11 loop cycle(s) in 00:00,090 minutes)
    -> 34 lexer rule(s), 424 parser rule(s)
INLINING RULES...
  FINISHED (4 rule(s) inlined and 47 reference(s) replaced in 00:00,231 minutes)
EXAMINING AMBIGUITIES...
changed Rules: bool_not_expr, vector_compare_bool_expr, math_sym, bool_constant,
neg_dir, var_datset_2, delete, var_equate_1, var_iterat_2, feat_def, var_geom_2,
var_geom_3, fly, var_units_2, var_units_3, var_close, var_eval_1, var_open_1,
var_output_2, var_output_3, var_output_5, var_read, var_macro_1, var_macro_2,
var_call_2, var_rmeas_2, var_rmeas_4, var_rmeas_5, var_rmeas_6, var_rmeas_7,
intrinsic_real_func, var_matdef_2, var_matdef_4, goto, calib_seq, var_sensor,
var_cmpntgrp, var_sensor_4, var_snsdef_3, var_snsdef_7, var_snsdef_9, var_snsgrp_1,
var_snsgrp_2, var_snslct_2, var_snslct_4, var_tol_2, var_tol_3, var_tol_4,
var_tol_5, var_tol_6, var_tol_13, var_tol_14, var_tol_15, var_tol_16, var_tol_17,
var_tol_18, var_tol_19, var_tol_20, var_decl_3, var_value_rt, var_value_bound,
var_value_snset
  FINISHED (62 rules changed in 00:00,080 minutes)
COLLECTING LEXER RULES TO PROTECT...
  FINISHED (10 rules found in 00:00,000 minutes)
REPLACING SINGLE CHARS IN PARSER RULES...
  FINISHED (61 lexer rule(s) generated and 1687 char(s) in 322 rule(s) replaced in
00:00,320 minutes)
SEARCHING FOR RESERVED WORDS...
  FINISHED (2 word(s) changed in 00:00,010 minutes)
notice: rule "WS" is never referred
notice: [rule name]([count referred by parser rule(s)], [count referred by lexer
rule(s)])
notice: rule "cart_coord"(31, 0) is equal to "vector"(63, 0)
notice: rule "obj_param"(2, 0) is equal to "algdef_param"(1, 0)
notice: rule "obj_param"(2, 0) is equal to "var_usetol_param"(2, 0)
notice: rule "var_line_2"(2, 0) is equal to "var_plane_2"(3, 0)
notice: rule "var_const_arc"(1, 0) is equal to "var_const_cparln"(1, 0)
notice: rule "var_const_arc"(1, 0) is equal to "var_const_ellips"(1, 0)
notice: rule "var_const_cparln"(1, 0) is equal to "var_const_ellips"(1, 0)
notice: rule "var_output_6"(1, 0) is equal to "var_wrist_4"(1, 0)
notice: rule "var_output_6"(1, 0) is equal to "var_value_sa"(1, 0)
notice: rule "algdef_param"(1, 0) is equal to "var_usetol_param"(2, 0)
notice: rule "var_calib_rotab_2"(1, 0) is equal to "var_rotab_3"(1, 0)
notice: rule "var_calib_rotab_3"(1, 0) is equal to "var_rotab_4"(1, 0)
notice: rule "var_cmpntgrp"(2, 0) is equal to "var_snsgrp_1"(2, 0)
notice: rule "var_snsdef_9"(1, 0) is equal to "var_snsgrp_2"(1, 0)
notice: rule "var_wrist_4"(1, 0) is equal to "var_value_sa"(1, 0)
FORMULATING ANTLR-GRAMMAR AND WRITING FILE "gen_TEST_input.g"...
  FINISHED (00:00,852 minutes)
GENERATING CLASSES...
  FINISHED (00:00,010 minutes)
GENERATING EBNF FILE...
  FINISHED (00:00,230 minutes)
GENERATING HTML-FILES...
  FINISHED (516 file(s) generated in 00:03,585 minutes)
RUNNING ANTLR...
1 warning(s) (lexical nondeterminism)
22 warning(s) (nondeterminism)
  FINISHED (00:02,654 minutes)
```

Anhang II - Protokoll der Änderungen an der DMI-Grammatik

lfd. Nr.	Anz. Warnungen			Änderungen
	lex.	synt.	ges.	
1	2136	372	2508	pk=1, lk=1, label_name und varname zu Lexerregeln gemacht, NOMINALS_LABEL, SENSOR_LABEL, TOLERANCE_LABEL, FEATURE_LABEL, DATUM_LABEL zu Parserregeln gemacht
2	1124	372	1496	Lexerregeln protected machen, wenn möglich
3	818	153	971	pk=2, lk=2
4	658	114	772	pk=3, lk=3
5	637	99	736	pk=4, lk=4
6	636	98	734	pk=5, lk=5
7	636	97	733	pk=6, lk=6
8	632	97	729	pk=7, lk=7
9	632	97	729	pk=10, lk=10
10	625	97	722	pk=7, lk=7, QISDEF_DEF zu Parserregel gemacht
11	600	98	698	nach Regeln gesucht, die Buchstaben und LABEL_NAME enthalten und im Lexer stehen: VAR_DISPLY_3, VAR_MACRO_2 und VAR_OPEN_3 zu Parserregeln gemacht
12	598	98	696	dito: VAR_RMEAS_3 zu Parserregel gemacht
13	584	98	682	gohome und jumpto zu Parserregeln gemacht
14	584	92	676	dmis_seq durch eigene Regel ersetzt (ohne [Links-]Rekursion)
15	584	92	676	comment durch eigene Regel ersetzt
16	499	92	591	comment_char_list in var_error_2 durch integer ersetzt
17	496	92	588	comment_char_list gelöscht
18	496	92	588	comment_char gelöscht
19	496	92	588	actuals_label auskommentiert
20	479	92	571	zero_val gelöscht
21	479	92	571	full_time und full_date gelöscht
22	155	115	270	string_val zu Parserregel gemacht
23	125	132	257	integer_val zu Parserregel gemacht
24	71	139	210	POSITIVE_INTEGER_VAL zu Parserregel gemacht
25	66	139	205	comp_dat zu Lexerregel gemacht
26	64	139	203	jumptarget durch jumpto_label ersetzt
27	59	139	198	var_eval_3 zu Parserregel gemacht
28	58	139	197	dat_Label zu Parserregel gemacht
29	50	139	189	eigene Regel für var_obtain_1 und var_obtain_2
30	24	129	153	eigene Regeln zur Zahlenerkennung: positive_integer, integer, real, positive_real, non_negative_real; NEGATIVE_DECIMAL, NEGATIVE_INTEGER, NEGATIVE_REAL, POSITIVE_DECIMAL gelöscht
31	17	129	146	NON_NEGATIVE_DECIMAL und NON_NEGATIVE_INTEGER gelöscht
32	14	126	140	ZERO_DECIMAL und ZERO_INTEGER gelöscht

lfd. Nr.	Anz. Warnungen			Änderungen
	lex.	synt.	ges.	
33	14	126	140	SIGN gelöscht
34	14	124	138	eigene Regel für deg_dms
35	12	123	135	eigene Regel für pct_real
36	12	123	135	DIGIT_LIST gelöscht
37	12	123	135	eigene Regel für version
38	15	111	126	positive_integer zu Lexerregel gemacht
39	10	111	121	VERSION und PCT_REAL gelöscht, positive_integer erw., Hierarchie
40	8	111	119	neg_dir zu Lexerregel gemacht, eigene Regel für var_trans_2, neue Regel TS_CHR45_PRBRD
41	8	111	119	ZERO_LIST gelöscht
42	8	111	119	rapid, var_fedrat_4, var_snset_1 geändert (pct_real_val statt pct_real)
43	8	111	119	math_sym zu Parserregel gemacht
44	5	121	126	Vorzeichenauswertung in Parser verschoben
45	3	121	124	eigene Regel für varname
46	2	121	123	VAR_TEXT_2 und VAR_TEXT_3 zu virtuellen Token gemacht und in varname manuell gesetzt
47	2	120	122	report_def mit greedy versehen (eigene Regel)
48	3	119	122	any_integer zu Lexerregel gemacht
49	3	92	95	var_usetol_param, var_value_sa, var_wrist_4, obj_param, var_output_6, var_call_param, algdef_param: generateAmbigWarnings=false
50	3	89	92	var_tol_17, var_tol_7, var_write geändert
51	3	89	92	cart_coord, pol_coord, vector geändert
52	3	82	85	var_sensor_7, geoalg_param: generateAmbigWarning=false
53	3	74	77	var_tol_1 geändert
54	3	73	76	var_snslct_5: generateAmbigWarning=false
55	3	71	74	var_scnset_2, var_snslct_3 geändert
56	3	68	71	var_snset_2 geändert
57	3	67	70	AmbigCheck01 eingeschaltet
58	2	67	69	eigene Regeln für compare_sym, bool_not_expr; neue Regeln: not, eq, ne, lt, le, gt, ge; bool_constant zu Lexerregel gemacht, bool_val zu Parserregel gemacht
59	2	56	58	eigene Regel für var_snslct_1, var_snsdef_7, var_sensor_5, var_vform
60	2	55	57	var_calib_3: generateAmbigWarning=false
61	2	41	43	eigene Regel für var_matdef_2, var_matdef_1, var_prompt_2, var_output_7, var_output_3, var_output_2, var_feat_1, var_disply_3
62	2	40	42	var_select_case geändert
63	2	36	38	deg_dms, cart_coord, pol_coord, vector: greedy=true
64	2	33	35	non_negative_integer, non_negative_real geändert
65	2	32	34	angle_expr korrigiert
66	1	54	55	eigene Regel für label_name, label_name zu Lexerregel gemacht, extended Terminal Replace abgeschaltet

lfd. Nr.	Anz. Warnungen			Änderungen
	lex.	synt.	ges.	
67	1	41	42	verbesserte Methode zur Mehrdeutigkeitsfindung; eigene Regel für POSITIVE_INTEGER; dmis_body entfernt; eigene Regel für ENDFIL; Exception-Handling eingebaut
68	1	40	41	comment aus dmis_stm entfernt
69	1	33	34	eigene Regel für dmis_def
70	1	32	33	eigene Regel für if_seq
71	1	31	32	dmis_seq: Prädikat bzgl. Iterat hinzugefügt
72	1	30	31	eigene Regel für angle
73	1	27	28	eigene Regel für var_datset_3
74	1	26	27	eigene Regel für var_iterat_3; var_iterat_4 gelöscht
75	1	25	26	eigene Regel für var_calib_2
76	1	24	25	eigene Regel var_snslct_5 geändert
77	0	0	0	datset_def geändert, eigene Regeln für Ausdrücke

ANHANG III - BEISPIEL EINER DMIS-INPUT-DATEI

```
01: $$ *****   Beispiel für eine fehlerhafte DMI-Datei   *****
02: DMISMN / 'A2_VW_STANDARD_DSE_3_5',04.0
03: CRSLCT/ALL
04: V(V1) = VFORM / ALL
05: DISPLY / TERM , DMIS, STOR , DMIS
06: FILNAM/'A2_VW_DEA_VMS__DSE_3_4.DMO'
07: UNITS  / MM,ANGDEC
08: MODE   / MAN
09: R(day) = REPORT/DATE
10: R(tim) = REPORT/TIME
11: $$ ausgabe datum zeit
12: OUTPUT/R(day)
13: OUTPUT/R(tim)
14: (ANFANG1)
15: DECL/INTGR, XX1
16: XX1=PROMPT/'Bitte eine Zahl eingeben (1-5)'
17: TEXT   / QUERY ,,(XX1),10, N,L,'.                        '
18: IF/((XX1.LE.0).OR.(XX1.GE.6))
19: TEXT   / OPER  ,'***** Bitte Zahlen von 1 - 5 eingeben *****  '
20: JUMPTO / (ANFANG1)
21: ENDIF
22: COUNT1=ASSIGN/  0
23: T(LX_PM001)  = TOL / CORTOL, XAXIS, -0.0100,  0.0100
24: T(LQDYPM0001)= TOL / DISTB , NOMINL,   26.78188,  -.0001,   .0001,YAXIS
25: T(Q141)=TOL/WIDTH, -.20000,  .20000
26: T(Q147)= TOL / DIAM  ,  -.0001,   .0001
27: T(L_FLAT_005)= TOL / FLAT,  0.05000
28: T(LSTRGHT005)= TOL / STRGHT,  0.05000,RFS
29: T(LANGPM002)= TOL / ANGL,   -0.02000,    0.02000
30: DECL/INTGR, COUNT1
31: TEXT   / OUTFIL,'APPRCH und RETRCT 10 mm                                    '
32: D(LM_NOM_MCS)=TRANS/XORIG,0.00000,YORIG,0.00000,ZORIG,0.00000
33: SAVE   /DA(LM_NOM_MCS)
34: CONST/POINT,F(L_MAN_P1MV),MOVEPT,FA(L_MAN_P1_Z),0.00000,0.00000,-10.00000
35: D(L_MAN_M1)=TRANS/XORIG,FA(L_MAN_P1MV),YORIG,FA(L_MAN_P1MV),ZORIG,FA(L_MAN_P1M)
36: F(F1111)=FEAT/PLANE,CART,0.0000,0.000,0.000,-0.02069821,-0.01745849,0.99963333
37: CONST/PLANE,F(F1111),BF,FA(L_MAN_P1MV),FA(L_MAN_P2MV),FA(L_MAN_P3MV)
38: F(L1)       = FEAT/ LINE,BND,CART,  -89.00000,   -9.00000,   -2.00000, $
39:                0.00000,0.00000,0.00000,-0.10061048,0.99492589,0.00000000
40: CONST  / LINE  , F(L1)       , BF, FA(L_MAN_P3MV), FA(L_MAN_P1MV)
41: F(F2222)=FEAT/PLANE,CART,0.00000,0.00000,0.00000,0.00000,1.00000,0.00000
42: DATDEF / FA(F1111)        , DAT(DD)
43: D(L_MAN___M5)= ROTATE / XAXIS,  -1.000
44: D(L_MAN___M6)= TRANS  / XORIG,0.00000,YORIG,0.00000,ZORIG,0.00000
45: COUNT1=ASSIGN/  COUNT1+1
46: F(LS1)= FEAT/ SPHERE, OUTER, CART,0.00000,0.00000,0.00000,-20.00000
47: MEAS   / SPHERE, F(LS1)       ,    5
48: GOTO   /     0.00000,    0.00000,   21.00000
49:  PTMEAS/CART,0.00000,0.00000,10.00000,0.00000000,0.00000000,1.00000000
50:  PTMEAS/CART,-9.79796,0.00000,2.00000,-0.97979590,0.00002916,0.20000000
51:  PTMEAS/CART,0.00000,-9.79796,2.00000,0.00000006,-0.97979586,0.20000017
52:  PTMEAS/CART,9.79796,0.00000,2.00000,0.97979590,0.00002916,0.20000000
53: GOTO   /     0.00000,   15.67673,   21.00000
54: ENDMES
55: F(LS1_XYZ)=FEAT/POINT,CART,0.00000,0.00000,0.00000,0.0000000,0.0000000,1.000000
56: CONST  / POINT , F(LS1_XYZ)     , TR    , FA(LS1)
57: OUTPUT / F(LS1_XYZ)     , T(LX_PM001)   , T(LY_PM001)   , T(LZ_PM001)
58: OUTPUT / FA(LS1_XYZ)    , TA(LX_PM001)  , TA(LY_PM001)  , TA(LZ_PM001)
59: LV_S1X_123456789A   =VALUE/TA(LX_PM001)  ,INTOL
60: ENDFIL
```

Fehler erkannt durch: Präprozessor, Parser, Semantik-Prüfer

ANHANG IV - PRÜFBERICHT DES DMIS-CHECKERS

```
01: DMIS Checker
02: checking Beispiel.dmi...
03:
04: preprocessing...
05: DMIS 04.0 DMI preprocessor
06: error: line 35 too long
07: error: line 55 too long
08: finished (60 line(s) in 00:00,040 minutes)
09:   2 error(s) found
10:
11: parsing...
12: DMIS 04.0 DMI parser
13: error: FILNAM: line 6:36: expecting TS_COMMA, found '
14: '
15: error: TEXT: line 17:17: unexpected token: ,
16: error: POSITIVE_REAL_VAL: line 46:59: unexpected token: -
17: error: line 59:18: varname too long!
18: finished (61 line(s) in 00:00,531 minutes)
19:   4 error(s) found
20:
21: semantic analyzing...
22: DMIS 04.0 DMI semantic analyzer
23: error: line 22:1: variable "COUNT1" used before declaration (-> line 30)!
24: error: line 59:1: variable "LV_S1X_123456789A" not declared!
25: finished (59 statement(s) in 00:00,010 minutes)
26:   2 error(s) found
```

Literatur- und Quellenverzeichnis

Aho, A. V./Sethi, R./Ullmann, J. D. [Compilerbau, 1999]: Compilerbau : Teil 1, 2. Aufl., München, Wien: Oldenbourg Wissenschaftsverlag, 1999

ANSI [Document Details, 2003]: ansidocstore: Product: 'Dimensional Measuring Interface Standard (ANSI/CAM-I 104.0)', 2003, http://webstore.ansi.org/ansidocstore/product.asp?sku=DMIS+4%2E0, Zugriff am 16.02.2004

ANSI/CAM-I (Hrsg.) [DMIS 04.0, 2001]: Dimensional Measuring Interface Standard 104.0-2001, Part 1, 2001

Baeumle, P./Alenfelder, H. [Compilerbau, 1995]: Compilerbau : Eine praxisorientierte Einführung, Hamburg: S + W Steuer- und Wirtschaftsverlag, 1995

DIN (Hrsg.), Hartlieb, B. (Bearb.) [Gesamtwirtschaftlicher Nutzen der Normung, 2000]: Gesamtwirtschaftlicher Nutzen der Normung, Berlin usw.: Beuth, 2000

Gamma, E. [Entwurfsmuster, 1996]: Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software, Bonn: Addison-Wesley-Longman, 1996

Holmes, J. [Object-Oriented Compiler Construction, 1995]: Object-Oriented Compiler Construction, New Jersey: Prentice Hall, 1995

Lesk, M. E./Schmidt, E. [Lex, o. J.]: Lex - A Lexical Analyzer Generator, o. J., http://dinosaur.compilertools.net/lex/index.html, Zugriff am 22.01.2004

Louden, K. C. [Compiler Construction, 1997]: Compiler Construction : Principles and Practice, Boston: PWS Publishing Company, 1997

Norvell, T. S. [The JavaCC FAQ, 2003]: The JavaCC FAQ (14.07.2003), 2003, http://www.engr.mun.ca/~theo/JavaCC-FAQ, Zugriff am 23.01.2004

o. A. [javacc: JavaCC Home, o. J.]: javacc: JavaCC Home, o. J., http://javacc.dev.java.net, Zugriff am 23.01.2004

o. A. [JavaCC: JJTree Reference Documentation, o. J.]: JavaCC: JJTree Reference Documentation, o. J., https://javacc.dev.java.net/doc/JJTree.html, Zugriff am 23.01.2004

Object Workshops [DMIS, 2003]: Object Workshops - DMIS, 2003, http://www.dmis.com/dmis.htm, Zugriff am 16.02.2004

Parr, T. [ANTLR Reference Manual, 2003]: ANTLR Reference Manual (19.01.2003), 2003, http://www.antlr.org/doc/index.html, Zugriff am 23.01.2004

Parr, T. [Why Use ANTLR?, o. J.]: Why Use ANTLR?, o. J., http://www.antlr.org/why.html, Zugriff am 23.01.2004

Terry, P. D. [Coco/R for Java, 2002]: Coco/R for Java (23.10.2002), 2002, http://www.scifac.ru.ac.za/coco/javacoco.htm, Zugriff am 23.01.2004

Terry, P. D. [Coco/R, 2003]: Coco/R compiler generator (07.08.2003), 2003, http://www.scifac.ru.ac.za/coco/, Zugriff am 26.01.2004

Volkswagen AG [Intranet : VW-Rundgang, o. J.]: Intranet : VW-Rundgang, o. J., http://mm2e.wob.vw.de/werk_wolfsburg/index_js.htm, Zugriff am 17.02.2004

Wilhelm, R./ Maurer, D. [Übersetzerbau, 1997]: Übersetzerbau : Theorie, Konstruktion, Generierung, 2. Auflage, Berlin, Heidelberg, usw.: Springer, 1997

Wocke, P. M. [Koordinatenmeßmaschinen, 1993]: Automatische Programmierung für Koordinatenmeßmaschinen basierend auf CAD-Daten : Generierung, Visualisierung und Modifizierung von Antastpunkten und Verfahrwegen, Düsseldorf: VDI Verlag, 1993