Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2005-15

# Bachelorarbeit

im Studiengang „Angewandte Informatik“

# Entwicklung eines Parsers für TTCN-3 Version 3 unter Verwendung des Parsergenerators ANTLR

Wei Zhao

am Institut für Informatik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen
1. August 2005

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel. +49 (5 51) 39-1 44 14
Fax +49 (5 51) 39-1 44 15
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 1. August 2005

Bachelorarbeit

# Entwicklung eines Parsers für TTCN-3 Version 3 unter Verwendung des Parsergenerators ANTLR

Wei Zhao

1. August 2005

Betreut durch Prof. Dr. phil.-nat. Jens Grabowski

Institut für Informatik

Georg-August-Universität Göttingen

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einleitung

TTCN-3 ist eine Sprache zur Beschreibung von Tests. Im Jahr 2000 wurde sie durch das Europäische Telekommunikationsstandardisierungsinstitut (ETSI) erstmals veröffentlicht. Die Entwicklung von Testlösungen mit TTCN-3 dient der Durchführung von Interoperabilitäts-, Skalierungs-, Konformitäts-, und funktionalen Tests. TTCN-3 bietet weiterhin eine Testplattform für Protokolle, Dienste, und Objekt und komponentenorientierte Systeme.

Für andere neue Techniken oder Protokolle wie IPv6, SIP, SCTP oder Web Services und CORBA kann TTCN-3 auch unterstützend und unmittelbar eingesetzt werden. Um den gestiegenen Markt- und Nutzererfordernissen zu entsprechen, wurde TTCN-3 weiterentwickelt.

Gegenstand dieser Arbeit ist die Entwicklung eines Parser-Tools für TTCN-3. Dieses Tool analysiert ein TTCN-3-Dokument auf lexikalischer und syntaktischer Ebene und stellt als Ergebnis einen Abstrakten Syntaxbaum (AST) zur Weiterverarbeitung zur Verfügung. Als Grundlagen der aktuellen Versionen (3.0) dienten die, in Erweiterter Backus-Naur-Form (EBNF) angegebene TTCN-3-Grammatik sowie ***ANTLR*** [3], ein Tool zur Generierung vom Parsern.

Die vorliegende Arbeit gibt zunächst eine kurze Einführung in die Sprache TTCN-3. Das zweite Kapitel behandelt die Funktionsweise der erwähnten Hilfstools. Im dritten Kapitel wendet sich die Arbeit dem Untersuchungsgegenstand der Implementierung und Weiterverarbeitung des Abstrakten Syntaxbaums zu, bevor im abschließenden vierten Kapitel vier eine Zusammenfassung und ein Ausblick gegeben werden.

# 2 Grundlagen

## 2.1 Die TTCN-3 Sprache

TTCN-3 [01] ist eine offene, flexible Programmierungssprache für den Systemtest. Bei der Entwicklung von TTCN-3 wurden nicht nur die Vorteile ihres Vorgängers TTCN-2 [01] übernommen, sondern auch die anderen starken Funktionen realisiert.

### 2.1.1 Grundkonzepte und Struktur

Der Test wurde abstrakt in TTCN-3 definiert und ist unabhängig von konkreten Implementierungen des zu testenden Systems und des Testsystems. Der Test wurde durch TTCN-3 Module und den darin enthaltenen Testfällen repräsentiert.

TTCN-3 stellt eine Kernsprache dar. Gleichzeitig erlaubt TTCN-3 die Anbindung verschiedener, auch graphischer Präsentationsformate an die Kernsprache. So werden im gegenwärtigen TTCN-3-Standard zwei Präsentationsformate unterstützt. Hierbei handelt es sich zum einen um das tabellarische Format und zum anderen um das graphische Format, welche die Reaktion von Test in Form von Sequenzdiagrammen visualisieren. Die beiden Formate sind abhängig von der Kernsprache und können nicht allein angewandt werden [07].

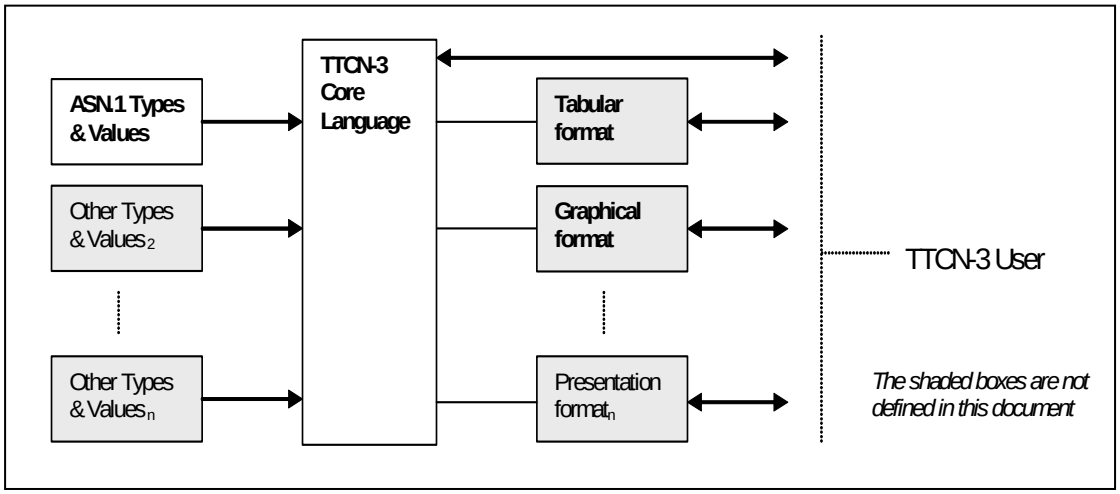


**Abbildung 2.1.1.1.0 Kernsprache und Formate von TTCN-3** [01]

## 2.1.2 Bestandteile der Kernsprache

In diesem Abschnitt werden einige ausgewählte Sprachelemente von TTCN-3 vorgestellt.

### 2.1.2.1 Module

Die Hauptkonstruktionen in TTCN-3 sind Module, Bestandteile eines Moduls sind Deklarationen von Konstanten, Typen, Daten, Funktionen, Testverhalten, Testfällen und die Kontrollfunktion. Um die Anpassung an spezielle Kontexte zu ermöglichen können die Module parametrisiert werden.

```
module ProtocolExample {
    import from externType all;
    const some;
    type record commonPacket {}
    template commonPacket packet :={}
    testcase tc()runs on mtcType system systemType{
       alt{} }
    function newInternetPTC(TestSystemType mySystem)
    runs on MtcType {}
    ……}
```

**Abbildung 2.1.2.1.1 Module Definition** [10]

Ein Modul besteht aus einem Definitionsteil und einem Kontrollteil.

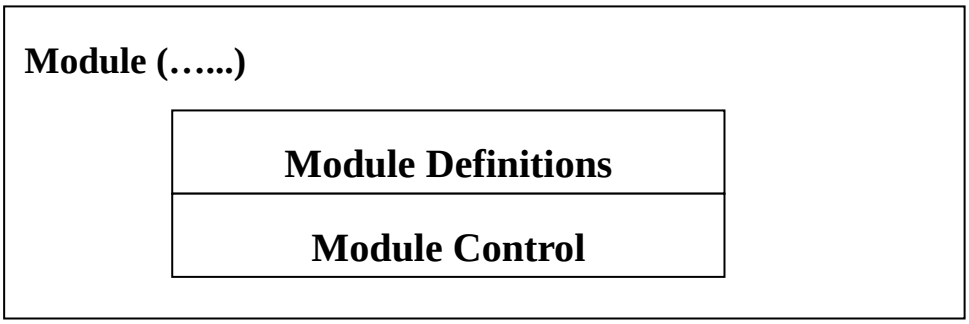


**Abbildung 2.1.1.1.2 Bestandteil von Module** [05]

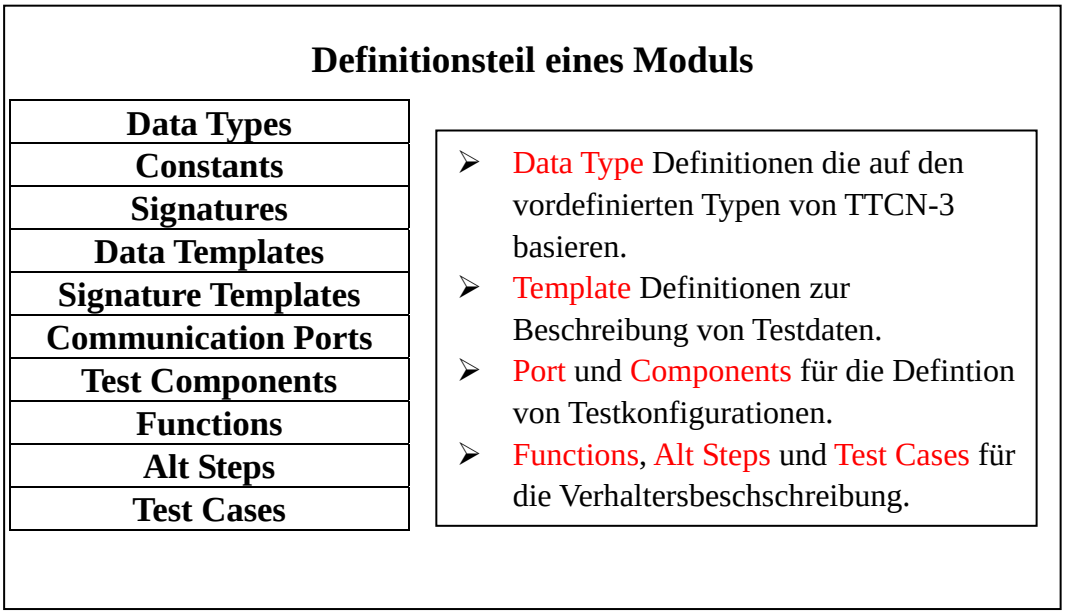


**Abbildung 2.1.1.1.3 Definitionsteil eines Moduls** [05]

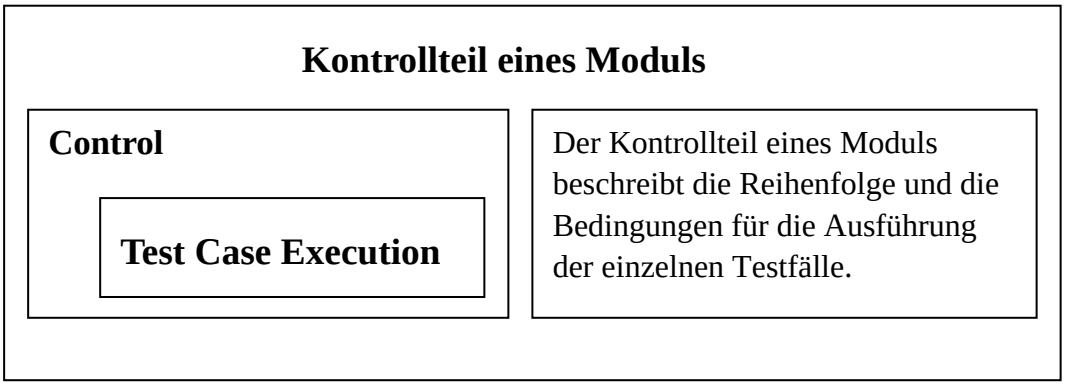


**Abbildung 2.1.1.1.4 Kontrollteil eines Moduls** [05]

### 2.1.2.2 Template

Eine *Template* ist eine Datenbeschreibung, in der, neben konkreten Werten, auch spezielle *Matching*-Operatoren zugelassen sind. Während der Testausführung wird geprüft, ob die von den Testkomponenten empfangenen Daten mit der zugehörigen *Template* konform sind.

In Abbildung 2.1.2.2.1, ist die *Template* BrachiosaurusTempate für den Datentyp dinosaurType, die vier Elemente definiert.

```
template dinosaurType BrachiosaurusTemplate := {
        Name      := "Brachiosaurus",
        Len       := ?,
        Mass      := ?,
        Time      := ?,
        place     := ?
 }
```

**Abbildung 2.1.2.2.1 template Definition** [11]

### 2.1.2.3 Testkonfigurationen

TTCN-3 erlaubt die dynamische Spezifikation von paralleler Konfiguration des Tests. Eine Konfiguration besteht aus einer Testkomponente, die über wohldefinierte ***Ports*** erfolgt und einer ***SUT*** (System unter Test). Jede Testkomponente und das ***SUT*** besitzen lokale Ports. Die Kommunikationsverbindungen zwischen den Testkomponenten und dem ***SUT*** werden durch das Verbinden der lokalen Ports erzeugt.

Abbildung 2.1.2.3.1 zeigt exemplarisch die Definition eines Ports für die Kommunikation mit den Lagern des ***SUT***.

```
type port httpPortType message {
   out   urlType;
   in   dinolistType;
   }
type component ptcType {
   port httpPortType httpPort;
   timer localTimer := 3.0;
   }
type component systemType {
   port httpPortType
   httpPortArray[NUMBER_OF_PTCS];
  }
```

**Abbildung 2.1.2.3.1 Port- und Komponent-Definitionen** [11]

Wie durch das Schlüsselwort **message** gekennzeichnet wird, ist ***httpPortType*** ein Port für eine nachrichtenbasierte Kommunikation. In TTCN-3 werden Nachrichten als Datentypen definiert. Prinzipiell kann hierbei jeder Wert eines Typs gesendet und empfangen werden. Über einen Port des Typs ***httpPortType*** können Werte des Typs ***urlType*** gesendet und Werte des Typs ***dinolistType*** empfangen werden. Testkomponenten besitzen lokale Ports und können zusätzlich noch lokale Timer, Konstanten und Variablen verwalten. Eine Testkomponente vom Typ ***ptcType*** besitzt den Timer ***localTimer*** und den Port ***httpPort***.

### 2.1.2.4 Testfälle

Ein Testfall ist ein speziale Funktion, die in **Module Control** durch ***execute*** gestartet wird. Das Resultat ausgeführter Tests ist ein Wert von ***verdicttype***. Jeder Testfall der im Funktionskörper enthaltenen Verhaltensbeschreibung spezifiziert das Verhalten der ***MTC*** (Main Test Component) [06].

Die Signatur des Testfalls besteht aus dem Testfallnamen, der Liste der formalen Parameter, einer Referenz auf den Typ der ***MTC*** (***runs on mtcType***) und einer Referenz auf den Typ des ***SUT*** (***system*** SystemType). Im Testfall werden zunächst die Testkomponenten erzeugt (***create***-Anweisungen), die neu erzeugten Testkomponenten mit anderen Testkomponenten und dem ***SUT*** verbunden (***map***-Anweisungen) und danach gestartet (***start***-Anweisung). Das Verhalten der einzelnen Testkomponenten wurde dabei als Referenz auf eine Funktionsdefinition (***ptcBehaviour***) übergeben.

```
testcase DinoListTest_1() runs on mtcType system systemType
 {
       var default def := activate(StandardDefault());
       var ptcType ptcArray[NUMBER_OF_PTCS];
       var integer i := 0;
       for (i:=0; i<NUMBER_OF_PTCS; i:=i+1) {
          ptcArray[i] := ptcType.create;
         }
       for (i:=0; i<NUMBER_OF_PTCS; i:=i+1) {
        map (ptcArray[i]:httpPort, system:httpPortArray[i]);
        }
       for (i:=0; i<NUMBER_OF_PTCS; i:=i+1) {
          ptcArray[i].start(ptcBehaviour());
       }
      for (i:=0; i<NUMBER_OF_PTCS; i:=i+1) {
       //warten auf  Ende des PTCs
       ptcArray[i].done;
      }
 }
```

**Abbildung 2.1.2.4.1 Testfall** [11]

### 2.1.2.5 Funktionen

TTCN-3 Funktionen dienen zum Ausdruck des Testverhaltens, zur Organisation der Ausführung von Test und zur Berechung eines Werts. Eine Funktion, die das Testverhalten spezifiziert, besitzt eine ***runs on***-Anweisung, die den Typ der Testkomponente angibt, auf der die Funktion ausgeführt wurde soll. Innerhalb der Funktion können die, im Komponententyp deklarierten Variablen, Konstanten, Timer und Ports benutzt wurden. So verwendet die in Abbildung 2.1.2.5.1, definierte

Funktion ***ptcBehaviour*** die in dem Komponententyp ***ptcType*** deklarierten Ports.

```
function ptcBehaviour() runs on ptcType {
     var default def := activate(StandardDefault());
      httpPort.send(urlTemplate);
      localTimer.start;
     alt       {
   [] httpPort.receive(DinoListTemplate) {
      localTimer.stop;
      setverdict(pass);
       }
   [] httpPort.receive {
      localTimer.stop;
      setverdict(fail);
        }
   [] localTimer.timeout {
      setverdict(fail);  }
}  }
```

**Abbildung 2.1.2.5.1 Funktion-Definition** [11]

### 2.1.2.6 AltStep

Ein ***StandardDefault*** Verhalten wurde in TTCN-3 durch ***AltStep*** definiert. ***AltStep*** bewirkte, dass eine Testkomponente mit einem negativen Testurteil terminiert, sobald ein Timer abläuft. Der ***AltStep*** wurde mit Hilfe der **activate**-Anweisung in Abbildung 2.1.2.5.1 als *Default-Verhalten* aktiviert und damit bei der Ausführung der ***MTC*** berücksichtigt.

```
altstep StandardDefault() runs on ptcType
  {
    [] httpPort.receive(charstring: ?)
    {
       httpPort.send(standardConversation);
       repeat;
    }
    [] any timer.timeout
    {
        setverdict(fail)
      }
    [] any port.receive
      {
        setverdict(inconc)
      }
 }
```

**Abbildung 2.1.2.6.1 AltStep-Definition** [12]

### 2.1.2.7 Testurteile

Zur Beurteilung von Testfällen bietet TTCN-3 den speziellen Datentyp ***verdicttype*** mit den Werten ***pass***, ***fail***, ***inconc*** und ***error*** an. Ein ***pass*** beschreibt den Fall, dass der Testzweck erreicht wurde. Das Testurteil ***fail*** bedeutet, dass das ***SUT*** aufgrund der Testfolge als fehlerhaft beurteilt werden kann. Das Urteil ***inconc*** beschreibt den Fall, dass der Testzweck nicht erreicht wurde, das ***SUT*** jedoch auch nicht fehlerhaft ist. Ein ***error*** beschreibt das Scheitern des Testfalls aufgrund eines Fehlers in den Testgeräten [06].

### 2.1.2.8 Module Control

Der Kontrollabschnitt eines Moduls dient i. d. R. dazu, verschiedene Testfälle (oder ***Altsteps***) auszuführen und ein Gesamturteil zu berechnen.

```
control
 {
    var verdicttype overallVerdict := pass;

   if (capabilityTesting and overallVerdict ==pass)
    {
     overallVerdict := basicCapabilityTests();
   }

 if (interworkingTesting and overallVerdict ==pass)
  {
  overallVerdict := serviceInterworkingTests();
  }
 /* Anruf des Tests */
 if (loadTesting and overallVerdict ==pass)
  {
  overallVerdict := loadTests();
 }

 if (qualityTesting and overallVerdict ==pass)
 {
  overallVerdict := qualityAssuranceTests();
 }
}
```

**Abbildung 2.1.2.8.1 Module Control Definition** [12]

## 2.2 Der Parsergenerator *ANTLR*

***ANTLR*** steht für *Another Tool for Language Recognition* [02], ANTLR ist eine Weiterentwicklung von *PCCTS*, einem Parsergenerator, der aus der Grammatik einer Programmiersprache sowohl ein komplettes Framework in Form von ***Parser***, ***Lexer*** als auch einem Syntaxbaumgenerator generieren kann.

Es unterstützt als Ausgabesprachen Java, C++ und C#. Dieses Framework erleichtert die Erstellung eines Parsers erheblich. Der Vorteil dieser Anwendung liegt darin, dass der Entwickler nur eine Grammatik definieren muss und ***ANTLR*** daraus den Source-Code für die gewünschte Programmiersprache selbstständig generiert. ***ANTLR*** ist ein frei verfügbarer, in Java implementierter Parsergenerator. Die aktuelle Versionsnummer lautet 2.7.5.

### 2.2.1 *ANTLR* Konzepte

***ANTLR*** akzeptiert EBNF-artige Grammatiken zur Beschreibung der lexikalischen Analyse und zur Sprachbeschreibung. Optional konstruiert ein erzeugter ***Parser*** zur Laufzeit automatisch einen abstrakten Syntaxbaum für erkannte Programme der Sprache. Deren Traverse kann mit einer weiteren Grammatik beschrieben wurden.

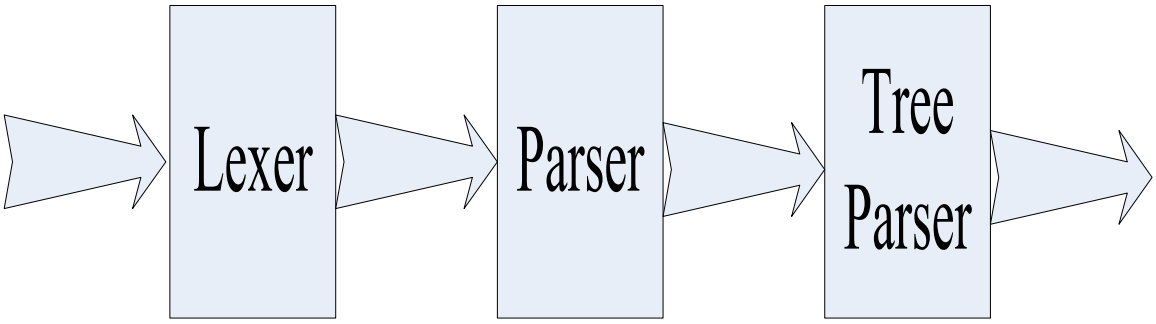


**Abbildung 2.2.1.1.0 Verarbeitungssphasen**

### 2.2.2 Lexer

Ein ***Lexer*** hat die Aufgabe, eine Folge von Eingabezeichen nach definierten Regeln in so genannte Token-Objekte umzuwandeln. Token-Objekte sind Instanzen der Klasse ***Token*** des ***ANTLR***-Frameworks und repräsentieren eine Folge von Zeichen, die eine syntaktische Einheit bilden. ***ANTLR*** erzeugt aus einer Beschreibung der lexikalischen Analyse in Form einer Grammatik eine Klasse, welche die spezifizierten Symbole erkennt. Die generierte Scanner-Klasse stammt fest von einer Klasse des Systems ab und adaptiert ein Interface als Schnittstelle zwischen einem ***ANTLR-Parser*** und der lexikalischen Analyse.

### 2.2.3 Parser

Der ***Parser*** organisiert Token-Objekten, die durch Parserregeln definiert sind. Die ***Parser***-Regeln entsprechen den Produktionsregeln einer Grammatik. Der ***Parser*** versteht sich auf *EBNF* und semantische Aktionen.

***ANTLR*** erzeugt ***Parser*** einer Klasse, dabei wird aus jeder Regel der Grammatik eine Methode im ***Parser*** angewendet.

### 2.2.4 Tree Parser

Der ***Tree Parser*** hat dieselbe Aufgabe wie ein ***Parser***. Allerdings durchsucht er einen Syntaxbaum von Token-Objekten, der durch den ***Parser*** aufgebaut werden kann, wenn dies in der Grammatik-Datei konfiguriert ist. Ein ***Tree Parser*** wird benötigt, wenn Berechnungen ausgeführt wurden müssen, die das Einlesen des kompletten Ausdrucks erfordern. In diesem Fall dient der Syntaxbaum als Zwischenform des durch den ***Parser*** interpretierten Ausdrucks.

### 2.2.5 Semantischen und syntaktischen Prädikate

***ANTLR*** erlaubt die Anwendung von Grammatiken, welche über eine zusätzliche Syntax mit semantischen und syntaktischen Prädikaten versehen sind, d.h. mit Nebenbedingungen, die erfüllt sein müssen, damit ein Zweig der Grammatik erkannt wird. Ein semantisches Prädikat ist ein Ausdruck in der Programmierungssprache wie Java oder C++. Ein syntaktisches Prädikat beschreibt eine Symbolfolge, welche der möglichen Erkennung des markierten Teils der Hintergrundgrammatik zu folgen hat.

### 2.2.6 Fehlerbehandlung

Während der Parsierung löst ein Fehler eine Exception aus. Jede Regelmethode generiert ***ANTLR*** automatisch einen Block, der derartige Exceptions abfängt, den Fehler berichtigt. Über eine weitere Syntax in der Grammatik können aber eigene Blöcke für verschiedenste Teile der Grammatik hinterlegt werden.

### 2.2.7 Beispiel von Anwendung für *ANTLR*

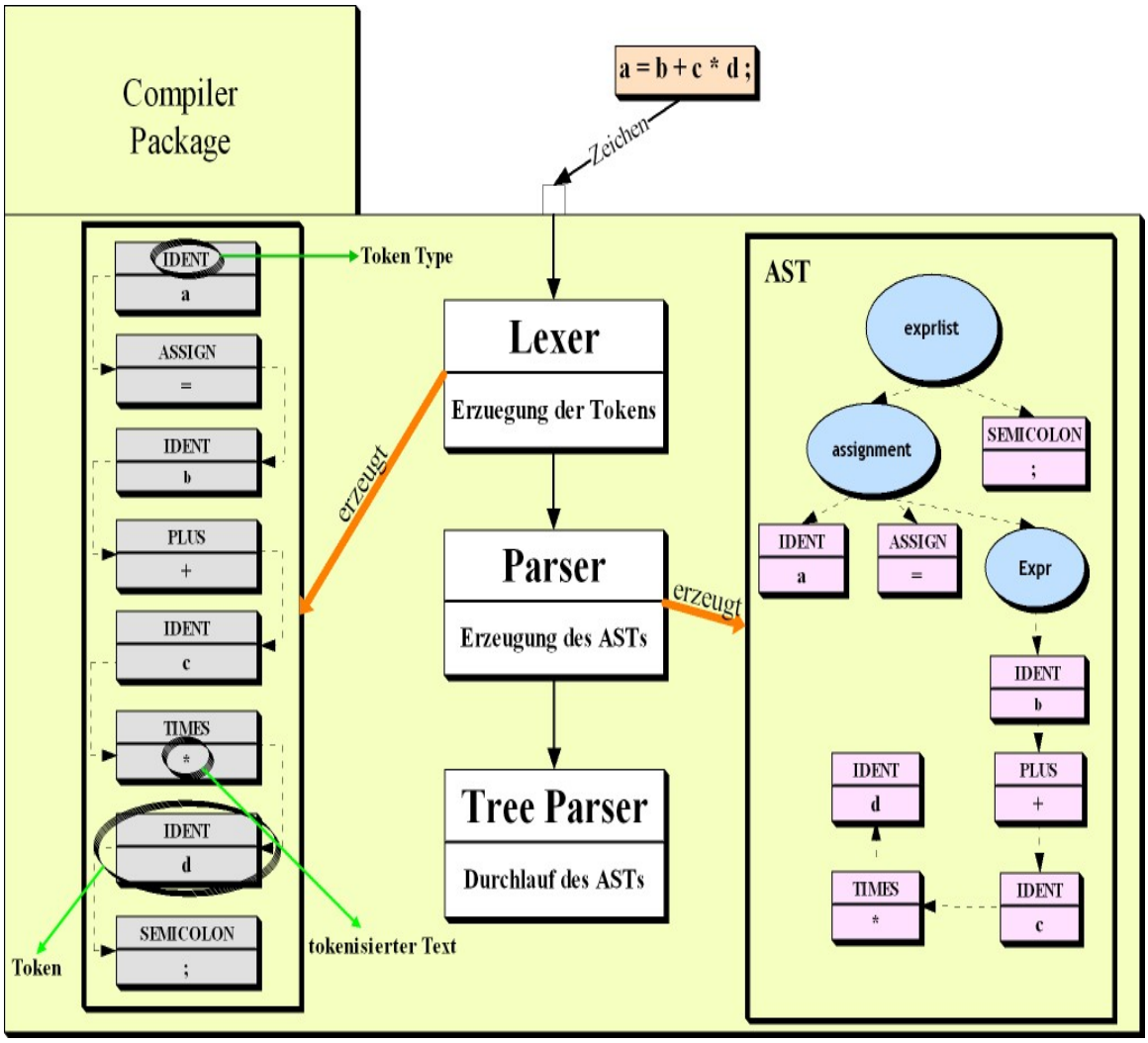


**Abbildung 2.2.7.1.0 Zeichenfolge -Tokenfolge - Abstrakter Syntaxbaum**

Dieser Abschnitt soll an Hand eines kleinen Beispiels einen Einblick in die Entwicklung eines ***Parser***s mit ***ANTLR*** geben. ***Parser*** bezieht sich in diesem Zusammenhang nicht auf die gleichnamige Verarbeitungsphase, sondern stellt vielmehr die Gesamtheit aller Prozesse in der Verarbeitungsphase dar, wie in

Abbildung 2.2.7.1.0 gezeigt wurde. Die Aufgabe des Beispielparsers soll es hierbei sein, eine Eingabe in Form

***a = b + c * d;***

zu erkennen, einen Abstrakten Syntaxbaum (AST) zu erzeugen und zu überprüfen, ob dieser AST einer bestimmten Form genügt.
Für das Beispiel sollen folgende Aufgaben gelöst wurden:
– Definition eines Scanner /Recognizer
– Symboltabelle erstellen
– Typüberprüfung
– Abstrakter Syntaxbaum
– Durchlaufen von Bäumen
– Schreiben eines Interpreters
– Testen des Output-Codes

## 2.2.8 Lexikalische Analyse

Mit dem in Abbildung 2.2.7.1.0 gezeigten Text als Eingabe erzeugt ***ANTLR*** einen ***Lexer***, der eine Zeichenfolge in eine Folge von ***Tokens*** überführt. Die möglichen Tokentypen bestehen aus Buchstaben, aus Operationszeichen (=, +, *) und aus Semikolon. Andere Zeichen sind in der Eingabe nicht zulässig.

## 2.2.9 *ANTLR*-Syntax

In Tabelle 2.2.9.1.1 wurden einige wichtige Syntaxelemente von ***ANTLR*** dargestellt. Die ausführlichen Funktionen der Elemente wurden im Abschnitt 2.2.11 erklärt

| | |
|---|---|
| **(.....)** | Unterregel |
| **(.....)*** | Unterregel kommt beliebig oft vor |
| **(.....)+** | Unterregel kommt mindestens einmal vor |
| **(.....)?** | Unterregel ist optional |
| **(.....)=>** | Semantische Aktion (Java oder C++) |
| **{.....}** | Syntaktisches Prädikat |
| **{.....}?** | Semantisches Prädikat |
| **\|** | Alternative |
| **:** | Regelanfang |
| **;** | Regelende |

| | |
|---|---|
| **!** | Verhindert Aufnahme des *Token*s in AST für Regel |
| ***#Label oder #id*** | AST des durch das ***id*** oder ***Label*** bezeichneten Elements |
| **##** | AST der aktuellen Regel, kurz für: ***#Regel-name*** |
| ***#[Token,"text"]*** | Erzeugung neuer Knoten mit Namen |
| ***#(root,id1,…,idn)*** | Erzeugung des Unterbaums mit ***root*** als Wurzel und ***id1*** bis ***idn*** als Kinder |

**Tabelle 2.2.9.1.1 *ANTLR* Syntaxelemente**

Alle ***Tokens***, die von dem Suffix „**!**" markiert werden, erscheinen nicht in dem erzeugenden AST. Im ***Parser*** und ***Tree Parser*** wurde ***ANTLR*** einige Aktionen von Benutzer übersetzen, um das AST leicht zu bauen. Beispielsweise wird eine Konstruktion, die mit „#" anfängt, übersetzt.

## 2.2.10 *ANTLR* Parser - Erzeugung eines Abstrakten Syntaxbaums

Die Erzeugung des AST erfolgt während des Parsing-Vorgangs. Zum Verständnis von Details muss zunächst auf die zur Darstellung des AST verwendete Datenstruktur eingegangen werden.

## 2.2.11 *ANTLR*-Eingabedateien für den BeispielParser

```
class BeispielLexer extends Lexer;
 options {
    k = 2; // Definition von lookhead
    exportVocab=BeispielLexer;
 }
 WS:(' '   // Definition von Leerzeichen
      |  '\t'
      |  '\n' // Definition von new line Zeichen
      |  '\r')
 ;
 IDENT  // Definition von Buchstaben und Nummer
 options {
       paraphrase = "identifier";
 }
   :  ('a'..'z' | 'A'..'Z' | '_' )
   ( ('a'..'z' | 'A'..'Z' | '_') |
   ('0'..'9' ))*
 ;
SEMICOLON // Definition von Semikolon
 options {
     paraphrase = ";";
 }
    : ';'
 ;
```

```
PLUS  // Definition von Operator „+“
 options {
       paraphrase = "+";
  }
   : '+'
 ;
 TIMES // Definition von Operator „*“
  options {
       paraphrase = "*";
 }
   : '*'
 ;
 ASSIGN // Definition von Operator „=“
  options {
        paraphrase = "=";
  }
   : '='
 ;
```

**Abbildung 2.11.1.1 Definition in BeispielLexer**

Die Gesamtheit der möglichen Zeichen, die in der Grammatik vorkommen, wurde im ***Lexer*** definiert.

```
class BeispielParser extends Parser;
options {
       language = Java;
           k = 2;
   importVocab=BeispielLexer;
   exportVocab=BeispielParser;
       buildAST = true;
}
exprlist
: ( assignmentStatement )*
   { ## = #([exprlist,"exprlist"], ##);};
```

In ***options*** wurde die Zielsprache, die Länge von lookahead, der Import-, der Exportwordschatz, und die Einstellung der Erzeugung von AST deklariert. Die letzte Zeile definiert die Aktion des Benutzers. Mit der Aktion ***exprlist*** wird ein neuer Knotenpunkt erzeugt, welcher den neuentstandenen AST als Zweig an die bestehende Struktur angliedert. In dem Fall, dass der AST leer war, bildet der ***exprlist*** Knoten die neue Wurzel.

Abbildung 2.11.1.2 zeigt die Definitionen von anderen Regeln in BeispielsParser.

```
assignmentStatement
 : assignment SEMICOLON!
 ;
assignment
 : (IDENT ASSIGN )? expr
```

```
 { ## = #([assignment,"assignment"], ##);};
primaryExpr
 : IDENT
 | expr
 ;
mulExpr
 : primaryExpr ( TIMES primaryExpr)*
  { ## = #([mulExpr,"mulExpr"], ##);};
expr
 : mulExpr ( PLUS mul_expr)*
 { ## = #([expr,"expr"], ##);};
```

**Abbildung 2.11.1.2 Definitionen in BeispielParser**

Weißt ein ***Token*** ein Suffix „!“ auf, so wurde die Baumkonstruktion dieses ***Tokens*** geschlossen, d.h., es kam nicht in dem AST vor. In Abbildung 2.11.1.2 kam ***Token*** „SEMICOLON“ nicht in dem AST vor.

```
class BeispielTreeParser extends TreeParer;
      option{
          buildAST = false;
         importVocab = BeispeilParser
      }
       exprlist  :
        #exprlist (assignment)*;
     //Deklaration möglicher Elemente von Kinder
       assignment :
        #assignment (IDENT ASSIGN)? expr;
       expr :
       #expr (mulExpr (PLUS mulExpr )*);
      mulExpr :
       #mulExpr (primaryExpr (TIMES primaryExpr)*);
      primaryExpr :
       #primary (IDENT|expr);
```

**Abbildung 2.11.1.3 Definitionen in BeispielTreeParser**

Diese Dateien wurden durch ***ANTLR*** die Zielsprache Java kompiliert.

## 2.2.12 Das Hauptprogramm des Parsers

***Lexer***, ***Parser*** und ***Tree Parser*** werden im Hauptprogramm konkretisiert. Im Hauptprogramm des BeispielParsers wurde eine BeispielLexer-Instanz erzeugt, die Zeichen aus dem Aufrufparameter des Programms ausließt. Weiterhin wurde ein BeispielParser konkretisiert, der seine ***Tokens*** wiederum von dieser BeispielLexer-Instanz bezieht. Der Parsing-Vorgang wurde durch den Aufruf, der als Methode der BeispielParser-Instanz implementierten Regel ***exprlist*** gestartet. Etwaige

Fehler in der Eingabe - ob auf lexikalischer oder syntaktischer Ebene - wurden im Rahmen der Ausnahmebehandlung gemeldet. Der, während des Parsing-Vorgangs erzeugte AST mittels der BeispielParser-Methode ***getAST*** vom ***Parser*** wurde abgeholt und in Textform ausgegeben. Anschließend wurde der BeispielTreeParser, dessen ***exprlist***-Regel aufgerufen wurde, konkretisiert um dadurch die Struktur des AST zu überprüfen. Diese Überprüfung konnte im gegebenen Beispiel nur dann fehlschlagen, wenn auf Grund eines Fehlers gar kein AST erzeugt wurde.

```
        import antlr.*;
        import antlr.collections.AST;
        import java.io.*;

        public class BeispielMain {
        public static void main (String argv[]) {
        BeispielParser beispielParser =
          new BeispielParser(new BeispielLexer(
                          new StringReader(argv[0])));
// Parser wurde konkretisiert.Lexer wurde als Parameter eingeben.
         try {
          BeispielParser.exprlist();
      // Ausführung von Parser
         } catch (RecognitionException exc)
      // Bearbeitung der Fehlung
              { System.err.println(exc); }
        if (beispielParser.getAST() != null)
        PrintAST(beispielParser.getAST(),0,
                 beispielParser.getTokenNames());
        BeispielTreeParser beispielTreeParser =
                                    new BeispielTreeParser();
// Tree Parser wurde konkretisiert
        try {
        beispielTreeParser.exprlist(BeispielParser.getAST());
// Der von Parser erzeugte AST wurde als Parameter eingeben.

         } catch (RecognitionException exc)
               { System.err.println(exc); }
        }

        public static void printAST (AST ast, int indent,
                                     String[] tokenNames) {
        for (int i = 0; i < indent; i++) System.out.print(" ");
          System.out.println(tokenNames[ast.getType()]+" \""
                                      +ast.getText()+"\"");
        for  (ast = ast.getFirstChild(); ast != null;
             ast = ast.getNextSibling())
          printAST(ast,indent+2,tokenNames);
// AST wurde auf dem Bildschirm gedruckt.
         }
        }
```

**Abbildung 2.2.12.1 BeispielMain.java**

Zum Test des Programms werden zwei Eingaben verglichen.

1. Korrekte Eingabe e=m+c*e;

```
$ java BeispielMain e=m+c*e;

IDENT "e"
ASSIGN "="
 IDENT "m"
 PLUS "+"
  IDENT "c"
  TIMES "*"
   IDENT "e"
```

Ein richtiger Text wurde als Parameter für das Hauptprogramm eingegeben, nach der Bearbeitung vom ***Parser*** wurde der erzeugte AST ausgegeben. ***IDENT "e"*** war die Wurzel des ASTs, und die anderen waren die Zweige.

2.Fehlerhafte Eingabe e=m+c*e ( Das schließende Semikolon fehlt.)

```
$ java BeispielMain e=m+c*e

line 1:8: expecting SEMICOLON, found 'null'
<AST>: expecting exprlist, found '<empty tree>'
```

Die falsche Eingabe würde die Fehlermeldung bekommen.

# 2.3 Das *ANT* System

## 2.3.1 Konzepte

***ANT*** steht für "Another Neat Tool", Das Programm ***ANT*** ist ein Werkzeug, welches für Java programmiert wurde und zum automatisierten Erzeugen von Programmen aus Quell-Code benutzt werden kann [04].

***ANT*** ist plattformunabhängig, solange auf der Zielplattform eine Java-Laufzeitumgebung (JRE) verfügbar ist. ***ANT*** wird durch eine XML-Datei gesteuert, die so genannte *Build-Datei*. In dieser Build-Datei wird ein *Projekt* definiert, welches *Targets* enthält. Diese sind vergleichbar mit Funktionen in Programmiersprachen und enthalten unter Anderem Aufrufe von *Tasks*. Ein *Task* ist ein untrennbarer Arbeitsschritt. Zwischen den *Targets* können und sollten Abhängigkeiten definiert wurden. Beim Aufrufen eines *Targets* löst ***ANT*** diese

Abhängigkeiten auf und arbeitet die *Targets* entsprechend ab[04].

Weiterhin ist ***ANT*** ein offenes System. Das bedeutet, dass *Tasks* beliebig erweitert werden können.

In der Tabelle 2.3.1.1 zeigt die benötige *Tasks*.

| Task Name | Beschreibung |
|---|---|
| ***javac*** | Kompilieren von Quell-Code. |
| ***copy*** | Kopieren von Dateien. |
| ***delete*** | Löschen von Dateien oder Verzeichnissen . |
| ***mkdir*** | Erzeugen von Verzeichnis |
| ***echo*** | Zeigen von Text |
| ***java*** | Ausführen von Java-Programm |
| ***antlr*** | Kompilieren ***ANTLR***-Synatax File zu Java-File |
| ***condition*** | Beurteilen von Bedingung (ob *TRUE* ist) |
| ***fail*** | Zeigen von Text (wenn *condition* false ist) |
| ***jar*** | Erzeugen von jar-File |

**Tabelle 2.3.1.1 Benötige Tasks von *ANT* im Regression-Test**

### 2.3.2 Beispiel für Automatische Erzeugung mit *ANT*

```
<project name="Beispiel build.xml" default="build" basedir=".">
   // Definition eines Project mit Namen " TTCN3 Parser"
  <property name="version" value="1.0"/>
  <property name="base" value="."/>
  <property name="java.source.dir" value="src"/>
  <property name="javac.dest" value="classes"/>
  <property name="lib" value="lib"/>
  <property name="jar.dest" value="jar"/>
  <property name="manifest.src" value="src/manifest.mf"/>
   // Definition von Attribute (Variable)

<target name="clean" description="Loeschung der Verzeichnisse">
  <delete dir="${javac.dest}/" />
  <delete dir="${jar.dest}/" />
</target>
 // Definition von Task Clean, um Verzeichnis zu löschen

<target name="init" description="Erstellung der Verzeichnisse ">
  <mkdir dir="${javac.dest}" />
  <mkdir dir="${jar.dest}" />
</target>
// Erstellung von Verzeichnis
```

```
<target name="build" depends="init" description="Kompilieren" >
  <javac srcdir="${java.source.dir}"
        destdir="${javac.dest}"
        debug="on">
     <classpath>
        <pathelement path="${classpath}"/>
          <fileset dir="${lib}">
            </fileset>
     </classpath>
  </javac>
</target>

    // Die Java-Source Code wurde unter eingestelltem CLASSPATH
    // kompiliert

<target name="jar" depends="build" description="Erzeugung von JAR
                                                File" >
  <copy file="${manifest.src}"
         tofile="${javac.dest}/manifest.mf" filtering="true"/>
  <jar jarfile="${jar.dest}/Beispiel.jar"
       basedir="${javac.dest}"
       manifest="${javac.dest}/manifest.mf" />
</target>
  // Erzeugung von JAR File


</project> //Ende
```

**Abbildung 2.3.2.1 Beispiel - Build.xml**

Das Beispiel besteht aus vier *Targets* und sieben *properties*. Jedes *Target* definiert eine Aufgabe und ***property*** deklariert den Pfad von Verzeichnis oder Information über das Projekt als Variable. Beispielsweise übernimmt *Target-**build*** die Aufgabe, dass die originale Java-code kompiliert und die Ergebnisse im Zielverzeichnis speichert werden. Die Attribute ***description*** jedes *Target* war eine Spezifikation für *Target*. Der Benutzer kann diese Spezifikation durch Parameter ***–p*** von ***ANT*** einfach lesen.

Einige *Targets* existieren nicht allein, sondern haben eine Abhängigkeitsbeziehung. Im *Target-**build*** wurde die Abhängigkeitsbeziehung als ein Attribut von ***depends*** definiert, d.h., *Target-**build*** ist abhängig von *Target-**init***, wobei *Target-**jar*** abhängig von *Target-**build*** ist.

# 3 Implementierung des Parsers für TTCN-3

Es besteht Unterschiede zwischen der zur Formulierung der Regeln verwendeten Syntax von TTCN-3 und der von ***ANTLR*** geforderten Syntax. Daher ist es notwendig, sowohl die TTCN-3-Syntax, als auch die Bearbeitung und die Prüfung des nach Parse entstandenen Ergebnisses entsprechend umzuformen. Zu diesen Zwecken müssen die folgenden Aufgaben implementiert werden:

- TTCN-3 Syntax in ANTLR-Syntax umschreiben
- Ein grafisches Hauptprogramm wurde durch den erzeugten AST nach Parse darstellt.
- Bei Syntaxfehler oder Parsefehler würde dies in einem besonderen Fenster die Fehlermeldung gezeigt.
- Durch die angebenden Parameter wurde der AST in einem File ausgegeben.
- Realisierung des Regression-Tests mit dem Hilfswerkzeug ***ANT***.

## 3.1 Parser

### 3.1.1 Gegenüberstellung von TTCN-3 Syntax in BNF und *ANTLR*-Syntax

Eine Zusammenfassung der wesentlichen Unterschiede zwischen den, im Standard verwendeten und den von ***ANTLR*** akzeptierten Schreibweisen findet sich in Tabelle 3.1.1.1.

| Bedeutung | Synatx in EBNF | *ANTLR* Syntax |
|---|---|---|
| **Regeldefinition** | **Regelname ::=** | **Regelname :** |
| **Ende einer Regeldefinition** | | **;** |
| **Verweis auf eine Regel** | **Regel** | **Regel** |
| **Optionaler Block** | **[...]** | **(...)?** |
| **Wiederholungsblock(>1)** | **{...}+** | **(...)+** |
| **Optionaler Wiederholungsblock** | **{...}** | **(...)*** |
| **Literal** | **“Text“** | **Text** |

**Tabelle 3.1.1.1 Gegenüberstellung Syntax in EBNF und *ANTLR*-Syntax**

### 3.1.2 Die Einstellungen der *ANTLR*

Die Einstellungen für Grammatik, File und Regel von ***ANTLR*** wurden am Anfang im Bereich von ***options*** definiert.

```
options
    {
                language  = Java;
       // Deklaration von generierender Sprache
              importVocab = TTCN3Lexer;
//Initialer Wortschatz wurde aus der Klasse-TTCN3Lexer
//importiert,und der Wert war abhängig von der Benutzersdefinition
              exportVocab = TTCN3Parser;
   //Der Wortschatz wurde zur Klasse-TTCN3Parser exportiert
                 buildAST = true;
  //Falls „true“, wurde Ein AST im Parser erzeugt.
     defaultErrorHandler = false;
  //Automatischer Fehlerfang wurde aktiviert.
              ASTLabelType = "LocationAST";
  // Deklaration von Labelstyp
                        k = 3;
  // Anzahl von lookahead, 3 für das Programm war passend.
    }
```

Die Tabelle 3.1.2.1 zeigt die notwendigen Optionen in ***ANTLR***

| Symbol | Wert | Beschreibung |
|---|---|---|
| **importVocab** | Benutzersdefinition | Initialer Wortschatz aus Grammatik-File |
| **exportVocab** | Benutzersdefinition | Exportierender Wortschatz in Grammatik-File |
| **buildAST** | true oder false | Erzeugung von AST in Parser |
| **defaultErrorHandler** | true oder false | Kontrollieren von Exception-handling |
| **ASTLabelType** | Benutzersdefinition | Spezifikation von definiertem Label |
| **language** | Java, C++ oder C# | Generierende Sprache |
| **k** | Ganzzahl<br>Benutzersdefinition | Tief von lookahead |

**Tabelle 3.1.2.1 Optionen in *ANTLR***

### 3.1.3 Umschreibung des TTCN-3 Syntaxs

Dieser Abschnitt erklärt die Arbeitsweise des BNF-Übersetzers am Beispiel der Regeln ***AltGuardList*** und ***StructDefBody*** aus den TTCN-3 BNF (Regelnr.541 und Regelnr.18):

```
AltGuardList ::= {GuardStatement
                | ElseStatement [SemiColon]}
```

```
StructDefBody ::= (StructTypeIdentifier
                    [StructDefFormalParList] |
                      AddressKeyword)
                  "{" [StructFieldDef {"," StructFieldDef}] "}"
```

Diese Regeln müssen in die folgende Form gebracht werden:

```
pr_AltGuardList:
       (
          ( pr_GuardStatement
            |
            pr_ElseStatement
           (pr_SemiColon!)?
          )*
       );

pr_StructDefBody:
 (
   (  (
pr_StructTypeIdentifier
        ( pr_StructDefFormalParList )?
       )|
        pr_AddressKeyword
  )
    pr_BeginChar!
    (
        pr_StructFieldDef
        (
            pr_Comma!
            pr_StructFieldDef
        )*
    )?
   pr_EndChar!
 )
```

Die Regeln wurden in TTCN3Parser umgeschrieben und die Namen der Regeln wurden ein Präfix „pr_“ eingefügt.

### 3.1.4 Nichtdeterministischer endlicher Automat

Während der Umschreibung von den TTCN-3 BNF wurde ein ***nichtdeterministischer endlicher Automat*** erzeugt, weil einige Regeln komplizierte Bestandteile enthielten.

***Ein nichtdeterministischer endlicher Automat*** kann für einen Zustand und ein gelesenes Eingabezeichen in mehrere Zustände übergehen, d.h. das Programm kann wegen der Mehrdeutigkeiten nicht (oder langsam) weiter erkannt wird. Das folgende Beispiel aus TTCN-3 BNF (Regelnr.13, Regelnr.15 und Regelnr.48) zeigt, wie die Mehrdeutigkeiten erzeugt wurden.

```
TypeDefBody ::= StructuredTypeDef | SubTypeDef
```

```
StructuredTypeDef ::= RecordDef | UnionDef |SetDef |RecordOfDef
                    | SetOfDef | EnumDef | PortDef |ComponentDef
SubTypeDef ::= Type (SubTypeIdentifier | AddressKeyword) [ArrayDef]
               [SubTypeSpec]
```

Wenn ***TypeDefBody*** erkannt wurde, gibt es zwei Möglichkeiten -***StructuredTypeDef*** oder ***SubTypeDef***. Aufgrund des komplizierten Bestandteils von ***StructuredTypeDef*** und ***SubTypeDef*** sind die Mehrdeutigkeiten einfach zu erstellen, was zu der Notwendigkeit führt, einen ***Nichtdeterministischen endlichen Automat*** zu erstellen.

Um die Verwendung eines ***Nichtdeterministischen endlichen Automaten*** zu vermeiden, ist es nötig, lediglich eine einzige Menge für den Bestandteil auszuweisen. ***ANTLR*** bietet Suffix „=>“ als Lösung hierfür an.

```
pr_TypeDefBody :
       (
          (   pr_RecordKeyword | pr_UnionKeyword |
              pr_SetKeyword    | pr_EnumKeyword |
              pr_PortKeyword   | pr_ComponentKeyword
           ) => pr_StructuredTypeDef
        | pr_SubTypeDef        )
```

Das Suffix „=>“ symbolisiert ein syntaktisches Prädikat, es spezifiziert eine Alternative, welche die Sprache benötigt um lookahead vorherzusagen.

Mit dem Suffix „=>“ kann man eine Bedingungsmenge für ***StructuredTypeDef*** definieren, d.h. ein Satz, bei welchem ***TypeDefBody*** den Inhalt in der Bedingungsmenge einschließt, wird als ***StructuredTypeDef*** erkannt, sonst als ***SubTypeDef***.

### 3.1.5 Erzeugung von *Token*

Die Deklaration jedes ***Tokens*** wurde in ***Lexer***-File TTCN3Lexer.g geschrieben. Die Abbildung 3.1.5.1 zeigte die partielle Deklaration in ***Lexer***-File.

```
tokens // Deklaration von Tokens
   {
     RecordKeyword; UnionKeyword; SetKeyword;    EnumKeyword;
     PortKeyword; ComponentKeyword; StructuredTypeDef;
     …
  }
IDENTIFIER // Deklaration eines identifier
options {
  testLiterals = true;
```

```
  paraphrase = "an identifer";
}
  :  ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
  ;
 …
```

**Abbildung 3.1.5.1 Deklarationen in TTCN3Lexer**

### 3.1.6 Erzeugung des ASTs

Durch zusätzliche Operation wurde ein ***Token*** als ein Knoten im AST erscheinen. Das folgende Beispiel zeigt die vollständige Definition und Operation der Regel ***StructuredTypeDef***.

```
pr_StructuredTypeDef{ LocationAST col = null;}:
       (
           a:pr_RecordDef { col = #a; }
           |
           b:pr_UnionDef { col = #b; }
           |
           c:pr_SetDef { col = #c; }
           |
           d:pr_RecordOfDef { col = #d; }
           |
           e:pr_SetOfDef { col = #e; }
           |
           f:pr_EnumDef { col = #f; }
           |
           g:pr_PortDef { col = #g; }
           |
           h:pr_ComponentDef { col = #h; }
       )
{
     ## = #([StructuredTypeDef,"StructuredTypeDef"], ##);
     if(col != null) ##.setLocation(col, col);
};
```

Dieses Beispiel enthält einige *Labels* (col, a:, b:, ...). Ein *Label* ist ein Bezeichner, der ein kleines Element symbolisiert. Bei dem Zugriff auf markierten Elementes wird ein Symbol „#“ vor dem *Label* eingefügt. Die Beschreibung von Symbol „##“ und „**#“** wurde bereits in der Tabelle 2.2.9.1.1 erläutert. Die Methode ***setLocation*** bestimmt die Position des ***Tokens***, d.h. die Position wird als eine Information im AST–Knoten dargestellt.

Der erzeugte AST kann:

- In einem grafischen Fenster dargestellt
- auf dem Bildschirm oder File ausgegeben

- als Parameter in TTCN3TreeParser weiter geleitet

werden. Die Abbildung 3.1.6.1 zeigt die Implementierung der Erzeugung des ASTs in der Klasse – ***iFrame***.

```
protected unterFrame FileParser(String fn, boolean isPrint) {

   unterFrame treeFrame = null; // Initialisierung von Unterfenster
   TTCN3Parser parser = null; // Initialisierung von Parser
    TTCN3Lexer lexer = null; //Initialisierung von Lexer
       boolean errorOccured = false;
       message.setText("Parsing ..........  \n ");

       try {
          lexer = new TTCN3Lexer(new DataInputStream(new
                                    FileInputStream(fn)));
// Eine Instanz von Lexer wurde durch eingegebene File erzeugt
          lexer.setTokenObjectClass("antlr.CommonToken");
          // Einstellung von Tokentyp
          lexer.setFilename(fn);
          parser = new TTCN3Parser(lexer);
        // Eine Instanz von Parser wurde erzeugt
          parser.setFilename(fn);
          parser.setASTNodeClass("LocationAST");
          parser.pr_TTCN3Module();
        // Ausführung von Parser
       } catch (NoViableAltException ex)
        {…}// Fehlersfang
((LocationAST)parser.getAST()).setVerboseStringConversion(true,
                                       parser.getTokenNames());
          ASTFactory factory = new ASTFactory();
          root = factory.create(parser.getAST());
          root = parser.getAST();

          treeFrame = new unterFrame(fn, root);
       // GUI von erzeugtem AST
          TTCN3TreeParser treeParser = new TTCN3TreeParser();
       // Eine Instanz von Tree Paser
          try {
             treeParser.pr_TTCN3Module(parser.getAST());
// Der erzeugte AST wurde durch Tree Parser weiter bearbeitet
          } catch (RecognitionException e) {
        message.setText("Tree parsing errors : " +
                           " " + e.getMessage());
        // Fehlermeldung wurde im Message-Fenster gezeigt
          }
       }  message.append(" " + fn + " " + "\n parse finished");
       lexer = null;
       parser = null;
         //Freisetzung von Lexer und Parser
       return treeFrame;
   }
```

**Abbildung 3.1.6.1 Implementierung der Erzeugung des ASTs**

### 3.1.7 Ausgabe der Regel in *ANTLR*-Tree-Parser-Syntax

Die TTCN3TreeParser-Instanz generiert schließlich die Ausgabe in einer Form, in der sie als Basis für die Definition eines Tree-Parsers geeignet ist. Abbildung 3.1.7.1 zeigt die Definition von Regel ***AltGuardList*** in TTCN3TreeParser.

```
AltGuardList :
        #(AltGuardList
          (
          ( GuardStatement
              |
            ElseStatement )*
           ));
```

**Abbildung 3.1.7.1 Definition von Tree Parser**

## 3.2 AST grafischer Darstellung

Ein abstrakter Syntaxbaum ist eine Datenstruktur. Die Benutzung einer graphischen Oberfläche erleichtert die Darstellung der Struktur von AST. Die Oberfläche ist ein Multi-Fenster Programm, bei welchem ein Hauptfenster mehrere Unterfenster enthalten kann. Jedes dieser Unterfenster ist ein AST, der in graphischer Art dargestellt wurde. Das Programm besteht aus drei Teilen, dem Menü, einem Message-Fenster und einem Unterfenster des graphischen ASTs.

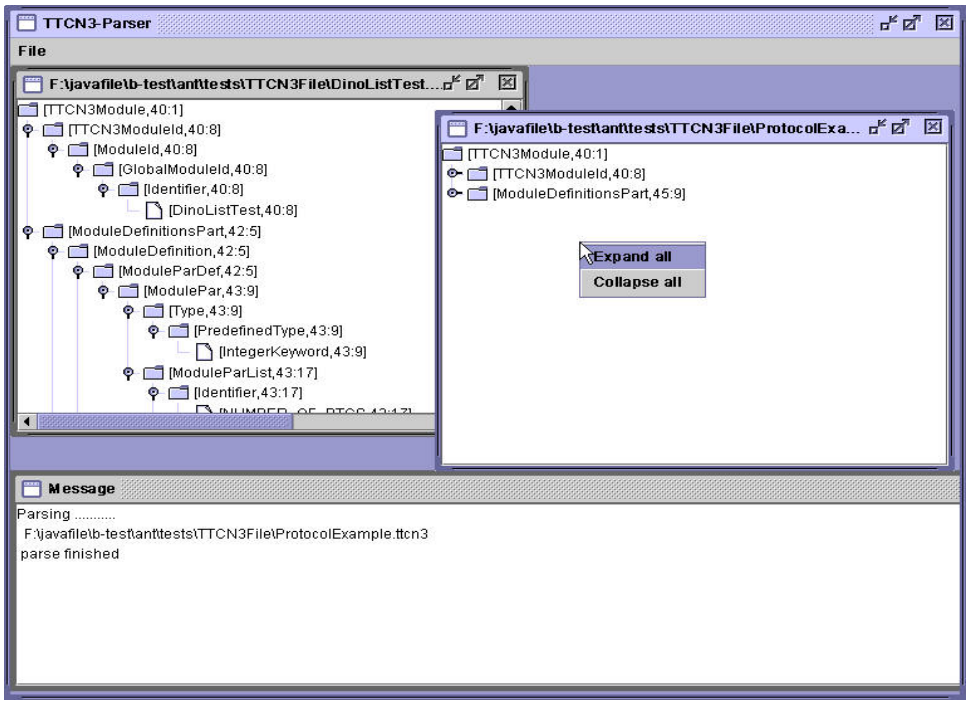


**Abbildung 3.2.1.0 Oberfläche von TTCN3-Parser**

### 3.2.1 Menü

Ein Benutzer kann in dem Menü (Item – *Open*) ein neues TTCN-3 Dokument öffnen oder das Programm schließen (Item - *Quit*). Jedes Menü-Item hatte eine eigene Schnelltaste, beispielsweise *Alt+N* für *Open*, um das Programm besser benutzen zu können.

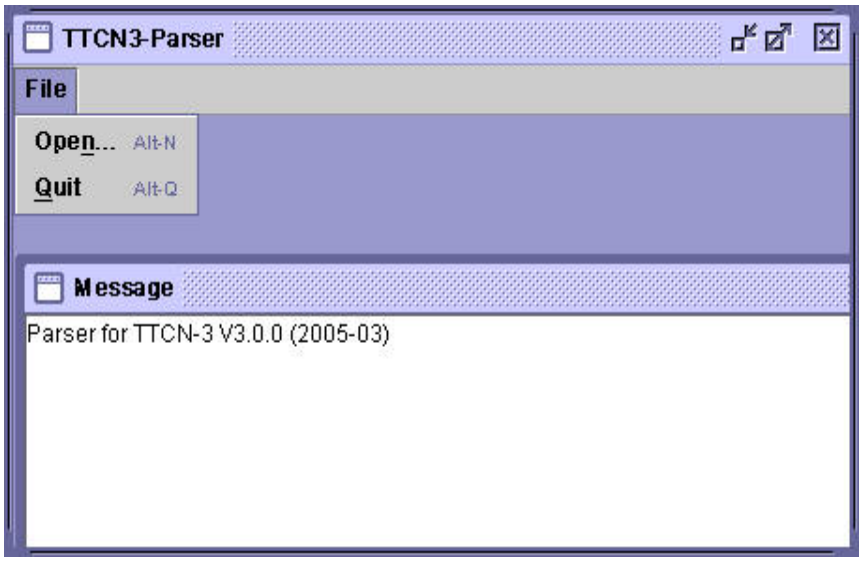


**Abbildung 3.2.1.1 Menü**

### 3.2.2 Message-Fenster

Die Informationen über Parser wurden im Message-Fenster gezeigt, um dem Benutzer den Zustand (Fehlermeldung oder Erfolgsmeldung) des Parsers zu zeigen.

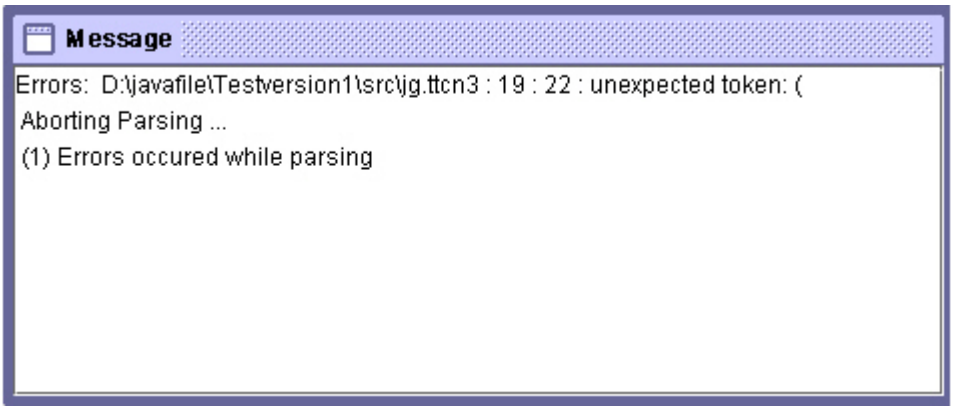


**Abbildung 3.2.1.3 Fenster für Information**

### 3.2.3 Unterfenster des ASTs

Nach dem Parser wurde ein AST von der Klasse ***TTCN3Parser*** erzeugt und seine ganze Struktur und alle Werte in dem Unterfenster gezeigt.

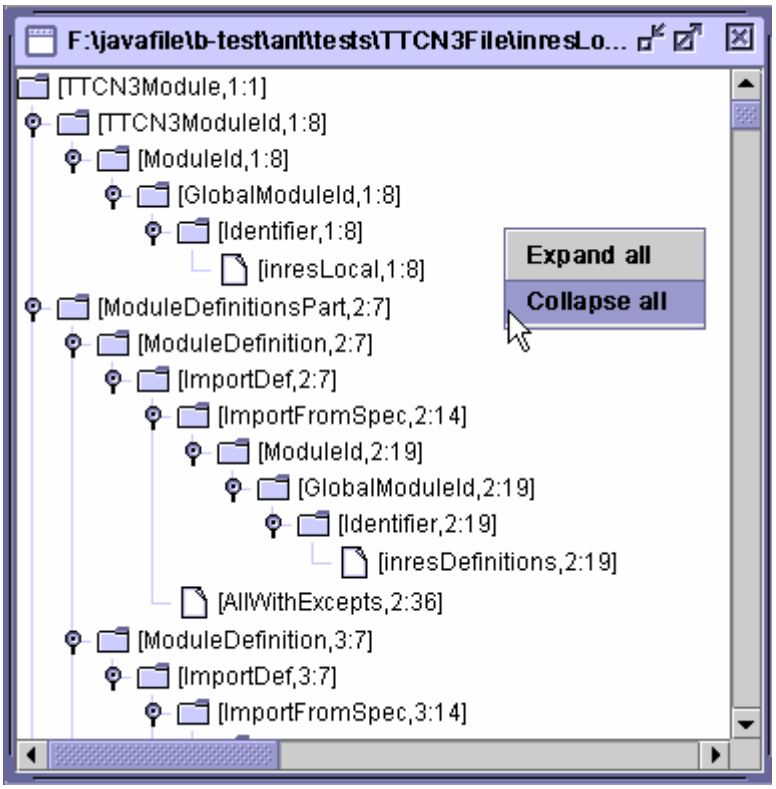


**Abbildung 3.2.1.4 Grafischer AST**

Der Benutzer kann durch die linke Maustaste das Funktionsmenü aufrufen, um den ganzen Baum zu öffnen oder zu schließen.

## 3.3 Ausgeben des ASTs

Um den AST weiter zu analysieren, kann der erzeugte AST durch die Parameter in **ASCII**-Format in einem Text-File gespeichert oder auf dem Bildschirm gedruckt werden.

Das Ausgeben des ASTs in ASCII-Format ist für den nächsten Schritt Regression-Test notwendig, denn ein Text-File im ASCII-Format ist sichtbar und vergleichbar. Abbildung 3.3.2.0 zeigt den Inhalt eines ASCII-Format Ausgabe-Files.

| Parameter | Bedeutung | Kommandozeileformat |
|---|---|---|
| **o** | Ausgabe des AST in ausgebendem File | **-o** *Filename DokumentList* |
| **p** | Ausgabe des AST in File **view.log** | **-p** *Dokumentlist* |
| **t** | Ausgabe des AST im Bildschirm | **-t** *Dokumentlist* |
| **v** | Ausgabe der Version | **-v** |
| **h** | Hilfe bei Eingabe des Parameters | **-h** |

**Tabelle 3.3.1 Programmsparameter**

**Beispiel:**

```
$ java TTCN3Main –t –o ast.log bd.ttcn3 eo.ttcn3
```

Abbildung 3.3.2.0 zeigt den Inhalt des ausgebenden Files.

```
****** parse tree of Testversion1/src/ProtocolExample3.ttcn3 *****

[TTCN3Module,40:1]
  [TTCN3ModuleId,40:8]
    [ModuleId,40:8]
      [GlobalModuleId,40:8]
        [Identifier,40:8]
          <[ProtocolExample,40:8]>
  [ModuleDefinitionsPart,45:9]
    [ModuleDefinition,45:9]
      [TypeDef,45:9]
        [StructuredTypeDef,45:14]
          [RecordDef,45:14]
            [Identifier,45:21]
              <[commonPacket,45:21]>
            [StructFieldDef,46:17]
              [Type,46:17]
                [PredefinedType,46:17]
                  <[OctetStringKeyword,46:17]>
              [Identifier,46:29]
                <[packetType,46:29]>
              [SubTypeSpec,46:40]
                [StringLength,46:40]
                  [SingleExpression,46:47]
                    [Primary,46:47]
                      [Value,46:47]
                        [PredefinedValue,46:47]
                          [Number,46:47]
```

**Abbildung 3.3.2.0 Ausgabe des erzeugten AST in ASCII Format**

## 3.4 Automatische Erzeugung mit *ANT*

Alle Schritte der Kompilierung originaler Codes können automatisch mit ***ANT*** durchgeführt werden. ***ANT*** steht außerdem ein spezielles *Target* für ***ANTLR*** zur Verfügung.

```
<target name="antlr" description=" Kompilieren antlr-Code">
  <antlr target="${java.source.dir}/TTCN3Lexer.g"
         outputdirectory="${java.source.dir}"/>
```

```
  <antlr target="${java.source.dir}/TTCN3Parser.g"
         outputdirectory="${java.source.dir}"/>
  <antlr target="${java.source.dir}/TTCN3TreeParser.g"
         outputdirectory="${java.source.dir}"/>
// Source Code wurde von ANTLR Kompiliert und in
// "outputdirectory" abgepeichert
</target>
```

**Abbildung 3.4.1.0 *ANTLR*-Target in *ANT***

Die Aufgabe von *Target-**antlr*** war Kompilierung der originalen Codes von ***ANTLR*** und Speicherung des Ergebnisses im Zielverzeichnis.

```
<target name="init" description="Erstellung der
                                 Verzeichnisse ">
     <mkdir dir="${javac.dest}" />
     <mkdir dir="${jar.dest}" />
</target>

<target name="build" depends="init,antlr"
        description="Kompilieren alle Java-code">
     <echo>
        Building... TTCN3 Parser Tools
     </echo>
     <javac srcdir="${java.source.dir}"
           destdir="${javac.dest}"
           debug="on">
       <classpath>
         <pathelement path="${classpath}"/>
           <fileset dir="${lib}">
             <include name="antlr.jar"/>
           </fileset>
       </classpath>
     </javac>
</target>
```

**Abbildung 3.4.2.0 Kompilierungstarget in *ANT***

Die Aufgabe von Target-***build*** war Kompilierung aller originalen Codes nach der Erzeugung des Ausgebensverzeichnis (*Target-**init***) und *Target-**antlr***.

## 3.5 Regression-Test

Nach der Korrektur eines Programms müssen Regression-Tests durchgeführt wurden. Dadurch wird sichergestellt, dass die Korrekturen richtig erfolgt sind und dass sie keinen Einfluss auf das richtige Funktionieren der unveränderten Teile des getesteten Programms haben. Dabei entfällt häufig, solange nur Fehler in der Implementierung

entfernt wurden, die Phase der Testdatengenerierung, da Testdaten schon von vorhergehenden Tests zur Verfügung stehen. Ein Verfahren für den automatischen Regression-Test dieses Programms wurde mit Hilfewerkzeug ***ANT*** angewandt. Die Abbildung 3.5.1.0 zeigt die Implementierung von Regression-Test mit ***ANT***.

```
<target name="createtest" depends="jar,deletelog"
        description="Erstellung des Testfile ">
  <java classname="TTCN3Main"  fork="true">
    <arg value="-o"/>
    <arg value="${log}/view.ref"/>
    <arg value="${ttf}/PizzaHutTest.ttcn3"/>
    <arg value="${ttf}/inresLocal.ttcn3"/>
    <classpath>
      <pathelement path="${classpath}"/>
        <fileset dir="${lib}">
          <include name="antlr.jar"/>
        </fileset>
        <fileset dir="${jar.dest}">
          <include name="ttcn3main.jar"/>
        </fileset>
    </classpath>
  </java>
</target> // Ausführung von TTCN3Main mit Parameter

<target name="runtest" depends="jar"
         description="Vergleichung der Testfiles ">
  <java classname="TTCN3Main"  fork="true">
    <arg value="-p"/>
    <arg value="${ttf}/PizzaHutTest.ttcn3"/>
    <arg value="${ttf}/inresLocal.ttcn3"/>
    <classpath>
      <pathelement path="${classpath}"/>
        <fileset dir="${lib}">
          <include name="antlr.jar"/>
        </fileset>
        <fileset dir="${jar.dest}">
          <include name="ttcn3main.jar"/>
        </fileset>
    </classpath>
  </java>
  <condition property="filesmatch">
    <filesmatch file1="${log}/view.log"
                file2="${log}/view.ref"/>
  </condition>
 <fail unless="filesmatch">
 The files view.log and view.ref do NOT match
   </fail>
    <echo>The files view.log and view.ref match</echo>
</target>//Regression-Test:Vergleichung von erzeugten files
```

**Abbildung 3.5.1.0 Implementierung von Regression-Test mit *ANT***

Die Aufgabe automatischer Erzeugung und Regression-Test wurde durch *Target*

***createtest*** und ***runtest*** durchgeführt.

*Target*-***createtest*** wurde in Beziehungsabhängkeit von *Target*-***jar*** und einem Befehl von Java als Source-Code kompiliert und ausführt. Nach der Ausführung wurde das Resultat in File ***view.ref*** abgespeichert. Im Anschluss wurde *Target*-***runtest*** ausgeführt, der Source-Code erneut kompiliert und dessen Ergebnis in File ***view.log*** abgespeichert.

Die *Task*-***condition*** hat verschiedene Funktionen, die abhängig von der Einstellung des Attributs-***property*** waren. Wenn der Wert von Attribut-***property filesmatch*** war, wurden zwei Files verglichen. Die Eingabe von Files wurde durch *Task*-***filesmatch*** realisiert. In diesem Beispiel wurden ***view.ref*** und ***view.log*** verglichen, das Ergebnis wird durch *Task*-***fail*** auf dem Bildschirm dargestellt.

Diese Art des Vergleichs wurde für File innerhalb von ***ANT*** implementiert. Alternativ wäre es möglich, einen manuellen Vergleich unter zur Hilfenahme von Vergleichssoftware, z.B. ***diff*** [09] vorzunehmen.

## 3.6 Probleme und Lösungen

### 3.6.1 JAVA - *String* und *StringBuffer*

Die Kopplungsoperator (+) von S*tring* ist ein einfaches Verfahren, mehre *Strings* zu einem längeren S*tring* zu vereinen. Die Ausgabe mit Hilfe des Kopplungsoperators ist in hohem Maße für ein kleines und festes *String-Objekt* geeignet. Dies gilt aber nicht für ein großes *String-Objekt*, da die Kopplungszeit n-Quadrat ist, wenn „n“ **Strings** mit dem Kopplungsoperator vereint werden.

Um die ganze AST am Bildschirm oder in dem File auszugeben, ist es notwendig, dass während des Durchlaufens eines ASTs der Wert jedes Knotens vereint wird.

```
                String ts = "";
protected boolean toStringList(LocationAST node, int level,
                                boolean isview)
 {
    LocationAST t = node;
    if (t.getFirstChild() != null) {
       ts = ts + t.toString() + "\n" ;
       // Vereinung des Werts
  if (isview){System.out.println(t.toString());}
    } else if (t != null) {
```

```
            Ts = "<" + t.toString() + ">" + "\n" ;
                  }
        if (t.getFirstChild() != null) {
            for (int i = 0; i <= level; i++) {
                ts = ts + "  " ;

      toStringList((LocationAST) t.getFirstChild(), level + 1,
                    isview);
           // Durch Rekursion wurde der nächster Knoten (Kind)
         //besucht.
        }
       if (t.getNextSibling() != null) {
            for (int i = 0; i < level; i++) {
               ts = ts + "  " ;
      toStringList((LocationAST)  t.getNextSibling(),  level,
                    isview);
        } // Durch Rekursion wurde der Knoten(Bruder) besucht.
        return true;
    }
```

**Abbildung 3.6.1.1 Method 1 - Kopplungsoperator**

```
          StringBuffer ts = new StringBuffer() ;
protected boolean toStringList(LocationAST node, int level,
                               boolean isview)
  {
           LocationAST t = node;
        if (t.getFirstChild() != null) {
            ts.append(t.toString() + "\n") ;
          } else if (t != null) {
            ts.append("<" + t.toString() + ">" + "\n") ;
                }
        if (t.getFirstChild() != null) {
            for (int i = 0; i <= level; i++) {
                ts.append("  ");

      toStringList((LocationAST) t.getFirstChild(), level + 1,
                    isview);
        }
        if (t.getNextSibling() != null) {
            for (int i = 0; i < level; i++) {
              ts.append("  ");

     toStringList((LocationAST) t.getNextSibling(), level,
                   isview);
        }
        return true;
  }
```

**Abbildung 3.6.1.2 Method 2 - StringBuffer**

Der Unterschied der Leistung zwischen den beiden oberen Methoden ist sehr groß. In der **Methode 2** wurde Variable ***ts*** vorher ein Speicherplatz erworben, der genug Raum für Ergebnis hat. Wenn Variable ***level*** 100 ist, ist **Methode 2** 90 mal schneller als **Methode 1**. Je länger ein *String* ist, desto größer ist der Unterschied der Leistung.

### 3.6.2 Neue Version von TTCN-3

Kürzlich wurde vom ETSI eine neue Version von TTCN-3 veröffentlich. Um TTCN-3 zu verbessern, wurden in der neuen Version einige Regeln in BNF geändert.
Die Änderung umfasst folgende Punkte:

1. Bestandteil der Regel wurde geändert.
2. Neue Regeln wurden eingefügt.
3. Einige Regeln wurden gelöscht oder ersetzt.

Die folgenden Beispiele zeigen die Unterschiede zwischen den beiden Versionen.

**In der Version V3.0.0Mockupv1 TTCN-3 BNF(2004-03) (Regelnr.: 3, 4und 5)**

```
TTCN3ModuleId ::= ModuleIdentifier [DefinitiveIdentifier]
ModuleIdentifier ::= Identifier
DefinitiveIdentifier ::= Dot ObjectIdentifierKeyword "{"
                         DefinitiveObjIdComponentList "}"
```

**In der Version V3.0.0 TTCN-3 BNF(2005-03) (Regelnr.: 3 und 4)**

```
TTCN3ModuleId ::= ModuleId
ModuleId ::= GlobalModuleId [LanguageSpec]
```

Die Regel ***ModuleIdentifier*** fand Eingang in die neue Version und wurde durch eine neue Regel ***ModuleId*** ergänzt.

Entscheidende Veränderung in den Versionen lassen sich z.B durch ***DiffDog***, eine spezielle Software für den Vergleich von ***Altova*** [08] auffinden. Nach dem Auffinden der Veränderungen wurden die entsprechenden Neuerungen in dieser Arbeit auf den Parser angewandt. Zusätzlich konnten zwei problematische Neuerungen herausgearbeitet werden. Die Regelnummer 305 ***CreateOp*** und Regelsnummer 338 ***KillTCStatement*** enthielten undefinierte Komponenten ***SingleExpression*** und ***ComponentIdentifierOrLiteral***.

# 4 Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein Parser für TTCN-3 Version 3 unter Verwendung des Parsergenerators ***ANTLR*** implementiert. Dazu wurde zu Beginn eine Einführung in die relevanten Grammatiken und Funktionen der TTCN-3 und des ***ANTLR*** gegeben.

Durch die folgenden Schritte wurde eine Parser für TTCN-3 implementiert.

1. Analyse des TTCN-3 Syntax und ***ANTLR*** Syntax
2. Umschreibung des TTCN-3 Syntax in ***ANTLR*** Syntax
3. Aufbau einer Oberfläche, um das Ergebnis des Parsers zu zeigen.
4. Ergänzungsfunktionen (inkl. Eingabe der TTCN-3 Dokument und Ausgabe auf dem Bildschirm oder in File )
5. Automatische Erzeugung und Regression-Test mit ***ANT***

Der Parser wurde anhand der aktuellsten Version von TTCN-3 (März 2005) implementiert. Der im Rahmen dieser Arbeit implementierte TTCN-3-Parser kann also als Grundlage für die Entwicklung weiterer, interessanter TTCN-3-Werkzeuge verwendet wurden.

Weitere Bearbeitungsmöglichkeiten, über den Untersuchungsgegenstand dieser Bachelorarbeit hinaus, sind evident, da sich die Weiterentwicklung zunächst nur mit dem Parser der kompletten Syntax von TTCN-3 beschäftigt. Die weiteren Bearbeitungsfelder könnten sich mit der Analyse von Semantik, und Herstellung der Symbol Tabelle zuwenden.

# Literaturverzeichnis


[01] European Telecommunications Standards Institute (ETSI), Final Draft ETSI ES 201 873-1V3.0.0 (2005-03): Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language (http://www.etsi.org/)

[02] Terrence Parr, Another Tool for Language Recognition (http://www.antlr.org/)

[03] The Apache ANT project (http://ant.apache.org/)

[04] Definition von ANT in lateinbord.de, Quelltext, WiKi (http://www.lateinboard.de/lexikon/Ant,bedeutung.htm)

[05] Jens Grabowski: Gedanken und Vorschläge zur Weiterentwicklung von TTCN-3, Vortrag, IST GmbH, Berlin, 11. Januar 2002.

[06] Jens Grabowski und Michael Schmitt : TTCN-3 Eine Sprache für die Spezifikation und Implementierung von Testfällen, 'at - Automatisierungstechnik', Oldenbourg Verlag, März 2002.

[07] Ina Schieferdecker: Eine TTCN-3 Testplattform für reaktive Systeme, Vortrag, Fraunhofer FOKUS/Technische Universität Berlin, 2004

[08] Altova diffdog (http://www.altova.com/products_diffdog.html)

[09] diff (http://www.lowfatlinux.com/linux-compare-files-diff.html)

[10] TTCN-3 Beispiel ProtocolExample.g, Testing Technologies, 2001-2005 (http://www.testingtech.de/ttcn-3_example)

[11] TTCN-3 Beispiel DinoListTest.g, Testing Technologies, 2001-2005 (http://www.testingtech.de/ttcn-3_example)

[12] TTCN-3 Beispiel PizzaHutTest.g, Testing Technologies, 2001-2005 (http://www.testingtech.de/ttcn-3_example)

## Anhang: Ergänzung/Korrektur

Einige Teile des in der Bachelorarbeit „Entwicklung eines Parsers für TTCN-3 Version 3 unter Verwendung des Parsergenerators ANTLR“ von Herrn Wei Zhao entwickelten TTCN-3 Parsers orientieren sich an folgender Open Source Quelle aus dem Internet:

Testing Technologies IST GmbH: *TTThreeparser.* Verfügbar unter der Adresse: http://packages.debian.org/unstable/devel/ttthreeparser [Aktualität: 09.01.2007]

Die Angabe dieser Quelle fehlt in der Arbeit von Herrn Wei Zhao.