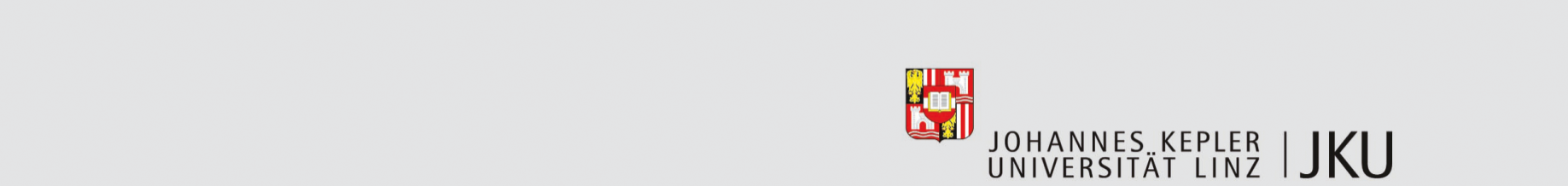
|  |  |
| --- | --- |
| TNF_Logo | Technisch-Naturwissenschaftliche  Fakultät |



**Eine VHDL Grammatik für Coco/R**

Bachelorarbeit

(Projektpraktikum)

zur Erlangung des akademischen Grades

Bakkalaureus der technischen Wissenschaften

im Bachelorstudium

Informatik

Eingereicht von:

Christian Reisinger, 0655859

Angefertigt am:

Institut für Systemsoftware

Beurteilung:

Prof. Dr. Dr. h.c. Hanspeter Mössenböck

Mitwirkung:

Dipl.-Ing. Markus Löberbauer

Linz, Oktober 2010

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bakkalaureatsarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und ilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Des weiteren versichere ich, dass ich diese Bakkalaureatsarbeit weder im In- noch im Ausland in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Bad Schallerbach, Oktober 2010 Christian Reisinger

Inhaltsverzeichniss

[1. Einleitung 4](#_Toc276901269)

[1.1. Einführung 4](#_Toc276901270)

[1.2. Aufbau 5](#_Toc276901271)

[2. LL(1) Konflikte und deren Behebung 6](#_Toc276901272)

[2.1. Faktorisierung 6](#_Toc276901273)

[2.2. Umwandlung in Iterationen 7](#_Toc276901274)

[2.3. Einsatz von Conflict Resolvers 8](#_Toc276901275)

[2.3.1. Konstante Anzahl von Lookahead Tokens 8](#_Toc276901276)

[2.3.2. Unbekannte Anzahl von Lookahead Tokens 9](#_Toc276901277)

[3. Scala AST Konstruktion 11](#_Toc276901278)

[3.1. Optionale Bestandteile 11](#_Toc276901279)

[3.2. Alternativen 12](#_Toc276901280)

[3.3. Iterationen 13](#_Toc276901281)

[3.4. Mehrere Rückgabewerte 13](#_Toc276901282)

[3.5. AST Beispiele 14](#_Toc276901283)

[3.6. AST Traversierung 16](#_Toc276901284)

[4. Zusammenfassung 17](#_Toc276901285)

[5. Literaturverzeichnis 18](#_Toc276901286)

# Einleitung

## Einführung

VHDL (Very High Speed Integrated Circuit Hardware Description Language oder auch VHSIC Hardware Description Language) ist eine vom IEEE[[1]](#footnote-1) standardisierte Sprache zur Beschreibung digitaler Schaltungen.

VDHL ist neben Verilog eine der weltweit am meisten genutzten Hardwarebeschreibungssprachen und hat sich in Europa zum "Quasi-Standard" entwickelt.

Die erste Spezifikation der Sprache wurde in den frühen 1980er Jahren entwickelt und ist das Ergebnis von Normierungsbestrebungen eines Komitees, in dem die meisten größeren CAD-Anbieter und CAD-Nutzer, aber auch Vereinigungen wie die IEEE, vertreten waren. Der größte nordamerikanische Anwender, das US-Verteidigungsministerium, hat VHDL zum Durchbruch verholfen, indem es die Einhaltung der Syntax von VHDL als notwendige Voraussetzung für die Erteilung von Aufträgen gemacht hat. [Ash02]

#### VHDL Beispiel eines D-Flip-Flop

**entity** DFlipflop **is**

**port**(D, Clk, Reset : **in** bit;

Q : **out** bit);

**end** DFlipflop;

**architecture** Rtl **of** DFlipflop **is**

**begin**

**process**(Clk, Reset)

**begin**

**if** Reset = '0' **then**

Q <= '0';

**elsif** Clk'event **and** Clk = '1' **then**

Q <= D;

**end** **if**;

**end** **process**;

**end** Rtl;

Im Rahmen dieser Arbeit wurde eine vollständige Grammatik des VHDL 2002 Standards für Coco/R[[2]](#footnote-2) entwickelt und es wird dargelegt, wie die dabei aufgetretenen Probleme gelöst wurden.

## Aufbau

Diese Bakkalaureatsarbeit untergliedert sich wie folgt:

* In Kapitel 2 wird eine Übersicht der möglichen LL(1) Konflikte und deren Behebung beschrieben.
* Kapitel 3 erörtert, wie diese Grammatik verwendet wurde, um einen abstrakten Syntax Baum zu erstellen.

# LL(1) Konflikte und deren Behebung

Die im Standard beschriebene Syntax liegt bereits in Extended Backus-Naur Form (EBNF) [Wirth77] vor, weist aber viele LL(1) Konflikte auf, die vorher behoben werden mussten.

In der Beschreibung von Coco/R [Möss03] werden drei verschiedene Möglichkeiten beschrieben, wie diese Konflikte gelöst werden können. Mit Hilfe deren ist es auch gelungen, die Grammatik in eine für Coco/R valide LL(1) Form umzuwandeln. Die drei Ansätze zur Behebung der Konflikte werden nachstehend mit Beispielen erörtert.

## Faktorisierung

Bei der Faktorisierung werden gemeinsame Teile herausgezogen und an den Anfang der Produktion gestellt. Z.B. kann folgende Produktion

A = a b c | a b d.

in eine Produktion A’

A’ = a b (c | d).

ohne Konflikte umgewandelt werden.

#### Grammatik Beispiel

Sequentielle Statements sind in VHDL folgendermaßen definiert:

SequentialStatement =

WaitStatement

| AssertionStatement

| ReportStatement

| IfStatement

| CaseStatement

| LoopStatement

| NextStatement

| ExitStatement

| ReturnStatement

| NullStatement

| SignalAssignmentStatement

| VariableAssignmentStatement

| ProcedureCallStatement

Auszugsweise werden hier nur die drei folgenden Produktionen gezeigt um den Konflikt deutlich zu machen:

NextStatement =

[Identifier colon] next [Identifier] [when Condition] semicolon.

ReturnStatement =

[Identifier colon] return [Expression] semicolon.

NullStatement =

[Identifier colon] null semicolon.

Man kann hier leicht den LL(1) Konflikt erkennen, da jede Alternative mit dem gleichen optionalen Nonterminal Identifier beginnt. Durch den Einsatz von Faktorisierung wurde der gemeinsame Teil herausgezogen und es ergeben sich damit nachstehende Produktionen:

SequentialStatement =

[**IF**(scanner.Peek().kind==\_colon) Identifier colon]

(WaitStatement

| AssertionStatement

| ReportStatement

| IfStatement

| CaseStatement

| LoopStatement

| NextStatement

| ExitStatement

| ReturnStatement

| NullStatement

| SignalAssignmentStatement

| VariableAssignmentStatement

| ProcedureCallStatement)

NextStatement = next [Identifier] [when Condition] semicolon.

ReturnStatement =return [Expression] semicolon.

NullStatement =null semicolon.

## Umwandlung in Iterationen

Linksrekursion stellt in LL(k) Sprachen im Gegensatz zu LR basierten immer ein Problem dar. In der Produktion

A = A b | c.

starten beide Alternativen mit c. Durch eine Umwandlung der Rekursion in eine Iteration kann dieses Problem gelöst werden, z.B wird die Produktion A zu

A‘ = c {b}.

#### Grammatik Beispiel

Name =

SimpleName

| OperatorSymbol

| SelectedName

| IndexedName

| SliceName

| AttributeName.

Prefix =

Name

| FunctionCall.

SliceName = Prefix ( DiscreteRange ).

Hier entsteht eine Linksrekursion, da Prefix wiederum ein Name sein kann, der wieder in einen SliceName abgeleitet werden kann. Durch die folgende Umwandlung in eine Iterationen wird diese Rekursion vermieden.

Name = NamePrefix {NamePart}.

NamePrefix =

Identifier

| stringLiteral.

NamePart =

NameSelectedPart

| NameAttributePart

| IF(isNameSlicePart())NameSlicePart

| NameAssociationListPart.

NameSlicePart = lparen DiscreteRange rparen.

## Einsatz von Conflict Resolvers

Bei dem Einsatz von Conflict Resolvers kann wiederum unterschieden werden, wie viele Tokens der Parser vorausschauen muss, um eine Entscheidung zu treffen.

### Konstante Anzahl von Lookahead Tokens

Diese Fälle könnten meistens auch durch den Einsatz von Faktorisierung gelöst werden, aber die Lesbarkeit der Grammatik würde darunter leiden. Durch die Berücksichtigung von mehreren Tokens in die Entscheidung verhält sich der Parser effektiv wie ein LL(k) basierter.

#### Grammatik Beispiel

LibraryUnit =

EntityDeclaration

| ArchitectureBody

| PackageBody

| PackageDeclaration

| ConfigurationDeclaration .

PackageBody =

package body Identifier is

{PackageBodyDeclarativeItem}

end [package body] [Identifier] semicolon.

PackageDeclaration =

package Identifier is

{PackageDeclarativeItem}

end [package] [Identifier] semicolon.

Der Konflikt kann gelöst werden indem k=2 Tokens berücksichtigt werden, dadurch ergibt sich folgende Lösungsmöglichkeit:

LibraryUnit =

EntityDeclaration

| ArchitectureBody

| **IF**(la.kind==\_package && scanner.Peek().kind==\_body)PackageBody

| PackageDeclaration

| ConfigurationDeclaration.

### Unbekannte Anzahl von Lookahead Tokens

Hier muss der Parser beliebig viele Tokens konsumieren und wird dabei in einen LL(\*) basierten umgewandelt. [Par07]

#### Grammatik Beispiel

InterfaceElementProcedure

InterfaceConstantDeclaration

| InterfaceVariableDeclaration

| InterfaceSignalDeclaration

| InterfaceFileDeclaration

InterfaceConstantDeclaration =

[constant] IdentifierList colon [in] SubtypeIndication [varAssign Expression].

InterfaceVariableDeclaration =

[variable] IdentifierList colon [InterfaceMode] SubtypeIndication [varAssign Expression].

Bei diesem Beispiel kann Coco/R ohne Hilfe nicht zwischen den letzten beiden Produktionen unterscheiden. Die Anzahl der benötigten Lookahead Tokens ist auch nicht konstant, da eine IdentifierList beliebig lang sein kann. Eine semantische Regel der Sprache definiert jetzt, wenn das Schlüsselwort in gefunden wurde, handelt es sich um eine InterfaceConstantDeclaration, sonst um eine InterfaceVariableDeclaration. Diese Information kann für folgenden Resolver genutzt werden:

**boolean** isInterfaceConstantDeclaration() {

**if** (la.kind==\_constant) **return** **true**;

**else** **if** (la.kind==\_variable || la.kind==\_signal || la.kind==\_file)

**return** **false**;

scanner.ResetPeek();

Token next;

**do** {

next = scanner.Peek();

}**while** (next.kind!=\_varAssign && next.kind!=\_semicolon

&& next.kind!=\_in);

**return** next.kind==\_in;

}

Bei dieser Funktion wird zuerst unterschieden, ob die Deklarations-Art explizit angegeben wurde. Wenn dies nicht der Fall ist, wird nach einem in Terminal vor einer Zuweisung oder einem Strichpunkt gesucht.

# Scala AST Konstruktion

Im folgenden Kapitel wird beschrieben, wie die Grammatik mit semantischen Aktionen erweitert wurde, um einen Abstrakten Syntax Baum zu erzeugen, und wie die dazugehörigen Scala Klassen definiert sind. Es wird dabei auch erklärt, wie die Grammatik durch möglichst streng typisierte Klassen beschreiben wird, um mit Hilfe des Scala Typesystems mögliche Fehler in weiteren Verarbeitungsschritten zu verhindern.

## Optionale Bestandteile

In Java wie auch in VHLD kann eine return-Anweisung einen optionalen Ausdruck enthalten:

ReturnStatement = return [Expression] semicolon

Diese Tatsache wird in vielen Java ASTs ignoriert, da Java nicht ausdrucksstark genug ist um dies zu Modellieren. Man kann dies im nächsten Code-Beispiel sehen, welches aus dem Java Compiler Tree API entnommen wurde[[3]](#footnote-3).

**public** **interface** ReturnTree **extends** StatementTree {

ExpressionTree getExpression();

}

Hier wird der Programmierer nicht daran gehindert ohne eine Überprüfung auf null den optionalen Ausdruck zu verwenden. Die vereinfachte und richtige Scala Version verwendet für optionale Ausdrückte das Option[[4]](#footnote-4) Monad, welches mit dem Haskell Maybe vergleichbar ist.

**final** **class** ReturnStatement(expression: Option[Expression])

Die dazugehörige vereinfachte Produktion von ReturnStatement verwendet die Hilfsfunktion toOption um eine Instanz von Scala Option zu erzeugen. Dies sieht folgendermaßen aus:

ReturnStatement<out ReturnStatement returnStmt > =

**(.**Expression expr=**null**;**.)**

return [Expression<out expr>] semicolon

**(.**returnStmt=**new** ReturnStatement(toOption(expr));**.)**.

Die erzeugten optionale Bestandteile der AST-Knoten können duch das in Scala vorhandene Pattern-Matching [Od08] sehr elegant verarbeitet werden, ohne das es dem Programmierer möglich ist, auf die notwendigen Überprüfung zu vergessen. Zusätzlich wird auch durch den Compiler überprüft, ob auch alle Fälle abgedeckt werder.

**returnStatement.expression match {**

**case None => //expression ist nicht vorhanden**

**case Some(expression) => //expression ist vorhanden**

**}**

## Alternativen

Alternativen haben wie auch der optionale Teil der Grammatik das Problem, dass man sie schlecht in Java beschreiben kann. Die einfachste Möglichkeit besteht wieder darin, dass die einzelnen Felder der AST Klassen null sein können, was wiederum keine saubere und typesichere Lösung darstellt. Hier bietet auch Scala mit der Either[[5]](#footnote-5) Klasse eine Lösung an.

VHDL definiert als Ziel einer Zuweisung eine Produktion Target welches zwei Alternativen hat:

Target =

Name

| Aggregate.

Diese Möglichkeiten lassen sich durch folgende Klasse typesicher beschreiben:

**final** **class** Target(**val** nameOrAggregate: Either[Name, Aggregate])

Die dazugehörende Produktion zur Erstellung des AST Knoten wird durch die in Java fehlende Typinferenz komplizierter.

Target<out Target target> =

**(.**target=**null**;**.)**

(

Name<out Name name>

**(.**target = **new** Target(**new** Left<Name, Aggregate>(name));**.)**

| Aggregate<out Aggregate aggregate>

**(.**target = **new** Target(**new** Right<Name, Aggregate>(aggregate));**.)**

).

Die erzeugten alternativen Bestandteile der AST-Knoten können vergleichbar wie die optionalen durch Pattern-Matching verarbeitet werden.

**target.nameOrAggregate match {**

**case Left(name) => //target ist ein name**

**case Right(aggregate) => //target ist ein aggregate**

**}**

## Iterationen

Wenn Nonterminale in einer EBNF Iteration verwendet werden, werden die einzelnen Elemente in einem Puffer gespeichert und als Typ wird Scala Seq[[6]](#footnote-6) verwendet.

Beispiel:

IdentifierList<out Seq<Identifier> list> =

**(.**ListBuffer<Identifier> identifierList=**new** ListBuffer<Identifier>();**.)**

Identifier<out Identifier identifier>**(.**identifierList.append(identifier);**.)**

{comma Identifier<out identifier> **(.**identifierList.append(identifier);**.)** }

**(.**list=identifierList.toList();**.)**.

## Mehrere Rückgabewerte

Um mehrere Werte zurück zugeben, werden die Scala TupleX Klassen verwendet, damit wird vermieden viele kleine Klassen zu implementieren, die nur Daten enthalten.

Beispiel:

EntityDesignator<out Tuple2<Identifier,Option<Signature>> designator> =

(. Identifier identifier=null; Signature signature=null;.)

(Identifier<out identifier>

|characterLiteral(.identifier=toIdentifier(t);.)

|stringLiteral(.identifier=toIdentifier(t);.)

) [Signature<out signature>]

(.designator=new Tuple2<Identifier,Option<Signature>>(identifier,toOption(signature)); .).

## AST Beispiele

Im folgenden Text wird anahnd von konkreten Beispielen genauer verdeutlich wie der konstruierte AST im Speicher dargestellt wird.

### WhileStatement

**while expr loop**

**statement1;**

**statement2;**

**end loop;**

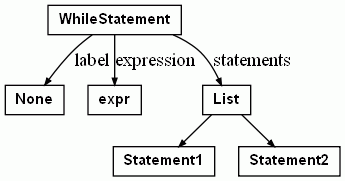


Abbildung 1: AST Knoten für ein While-Statement

### IfStatement

**if expr1 then**

**statement1;**

**elsif expr2 then**

**statement2;**

**else**

**statment3;**

**end if;**

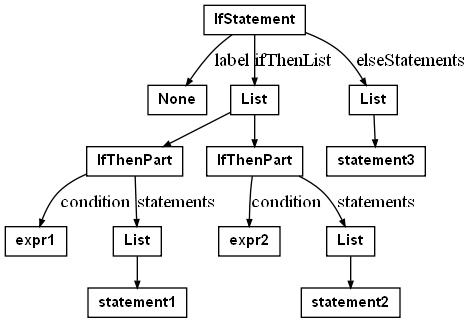


Abbildung 2: AST Knoten für ein If-Statement

### LogicalExpression

**a + b sll 1 <= c\*d and not e**

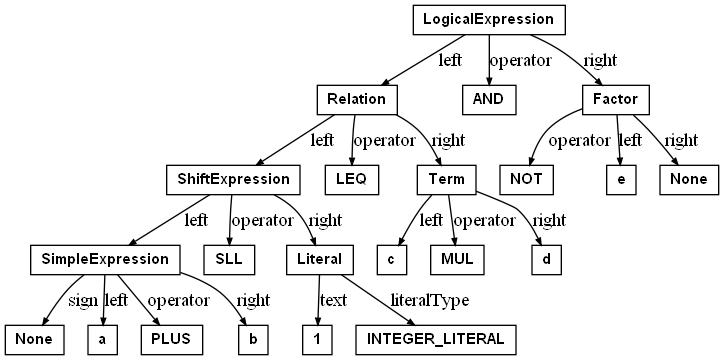


Abbildung 3: AST Knoten für eine Logical-Expression

### ConstantDeclaration

## AST Traversierung

Scala ermöglich es duch sein elegantes Pattern-Matching System den erzeugen AST sehr einfach mit einem rekursiven Alogirthmus zu traversieren:

**def** acceptNode(node: ASTNode): **Unit** = {

**def** acceptList[T <: ASTNode](list: Seq[T]): **Unit** =

list.foreach(acceptNode)

node **match** {

**case** **null** =>

**case** DesignFile(designUnits) => acceptList(designUnits)

**case** unit: DesignUnit => unit.libraryUnit.foreach(acceptNode)

**case** packageBodyDeclaration: PackageBodyDeclaration =>

acceptList(packageBodyDeclaration.declarativeItems)

**case** packageDeclaration: PackageDeclaration =>

acceptList(packageDeclaration.declarativeItems)

**case** entityDeclaration: EntityDeclaration =>

acceptList(entityDeclaration.declarativeItems)

acceptList(entityDeclaration.concurrentStatements)

**case** functionDefinition: FunctionDefinition =>

acceptList(functionDefinition.declarativeItems)

acceptList(functionDefinition.sequentialStatementList)

**//… weitere AST Knoten**

}

}

Scala ermöglicht es auch hier, duch sein elegantes Pattern-Matching System den erzeugen AST sehr einfach mit einem rekursiven Alogirthmus zu traversieren.

# Zusammenfassung

Im Rahmen dieser Arbeit wurde eine VHDL Grammatik für Coco/R erstellt. Es wurde dabei erklärt, wie die vorhandenen LL(1) Konflikte mit drei Strategien gelöst wurden. Weiters wurde dargelegt, wie diese Grammatik verwendet wurde, um einen AST erzeugen. Es wurden dann anhand von verschieden Code Beispielen gezeigt, wie dieser gespeichert wird, anschließen wurde verdeutlicht wie der AST traverstiert werden kann, um verschiedene Eigenschaften zu überprüfen.

# Literaturverzeichnis

[IEEE02] IEEE: Standard VHDL Language Reference Manual, IEEE Std 1076™-2002

[Möss03] Mössenböck, H.: The Compiler Generator Coco/R, User Manual

[Par07] Terence Parr: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX 2007.

[Ash02] Peter J. Ashenden: The Designer's Guide to VHDL. Morgan Kaufmann Publishers, San Francisco 2002

[Wirth77] Wirth, N.: What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions? Communications of the ACM, November 1977

[Od08] Martin Odersky, Lex Spoon, Bill Venners: Programming in Scala

# Abbildungsverzeichnis

[Abbildung 1: AST Knoten für ein While-Statement 14](#_Toc285547025)

[Abbildung 2: AST Knoten für ein If-Statement 15](#_Toc285547026)

[Abbildung 3: AST Knoten für eine Logical-Expression 15](#_Toc285547027)

a) wo weicht die atg von der Spezifikation ab? nirgends

b) warum gibt es welche Resolver?

c) wie ist der ast aufgebaut? Evtl. anhand von kleinen Beispielen. done

d) was wird wie am ast geprueft?

e) so viel von scala zeigen, dass man versteht wie der ast-Durchlauf

funktioniert. Das bitte nebenbei zeigen, zB in "Kasterln" wie man es in

Buechern haeufig findet. done

1. www.ieee.org [↑](#footnote-ref-1)
2. http://ssw.jku.at/Research/Projects/#Coco [↑](#footnote-ref-2)
3. http://download.java.net/jdk7/docs/jdk/api/javac/tree/com/sun/source/tree/ReturnTree.html [↑](#footnote-ref-3)
4. http://www.scala-lang.org/api/2.8.0/scala/Option.html [↑](#footnote-ref-4)
5. http://www.scala-lang.org/api/2.8.0/scala/Either.html [↑](#footnote-ref-5)
6. http://www.scala-lang.org/api/2.8.0/scala/collection/Seq.html [↑](#footnote-ref-6)