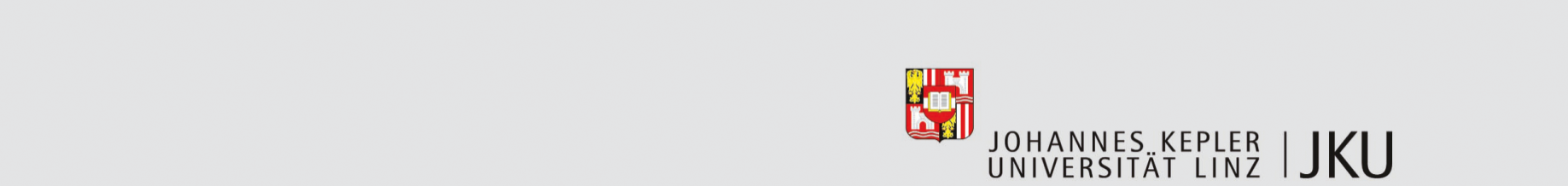
|  |  |
| --- | --- |
| TNF_Logo | Technisch-Naturwissenschaftliche  Fakultät |



**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:

Dipl.-Ing. Markus Löberbauer

Linz, Februar 2011

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, Februar 2011 Christian Reisinger

Inhaltsverzeichnis

[1. Einleitung und Motivation 4](#_Toc286262875)

[1.1. Einführung 4](#_Toc286262876)

[1.2. Aufbau 5](#_Toc286262877)

[2. LL(1) Konflikte und deren Beseitigung 6](#_Toc286262878)

[2.1. Faktorisierung 6](#_Toc286262879)

[2.2. Umwandlung in Iterationen 7](#_Toc286262880)

[2.3. Einsatz von Conflict Resolvers 9](#_Toc286262881)

[3. Scala AST Konstruktion 11](#_Toc286262882)

[3.1. Optionale Bestandteile 11](#_Toc286262883)

[3.2. Alternativen 12](#_Toc286262884)

[3.3. Iterationen 13](#_Toc286262885)

[3.4. Mehrere Rückgabewerte 13](#_Toc286262886)

[3.5. AST Hierarchie 14](#_Toc286262887)

[3.6. AST Beispiele 15](#_Toc286262888)

[3.7. AST Traversierung 17](#_Toc286262889)

[4. Beispiel-Anwendung 19](#_Toc286262890)

[4.1. Einleitung 19](#_Toc286262891)

[4.2. Implementierung 20](#_Toc286262892)

[5. Zusammenfassung 21](#_Toc286262893)

[Literaturverzeichnis 22](#_Toc286262894)

[Abbildungsverzeichnis 22](#_Toc286262895)

# Einleitung und Motivation

## Einführung

Coco/R ist ein am Institut für Systemsoftware an der Johannes Kepler Universität entwickelter Compiler Generator, welcher aus einer Sprachbeschreibung einen LL(1) Parser und einen Scanner erzeugt. [Möss03]

Die Beschreibung der Sprache muss in Form einer Attributierten Grammatik (ATG) angegeben werden. Eine ATG ist eine Beschreibung, die alle lexikalischen und syntaktischen Regeln der Sprache enthält.

Coco/R gibt es in Varianten für unterschiedliche Programmiersprachen wie beispielsweise C#, Java, C++ und Delphi. Um Coco/R bequem nutzen zu können, gibt es Erweiterungen für die Entwicklungsumgebungen Visual Studio, Eclipse und NetBeans. In dieser Arbeit wurde die Java Version von Coco/R verwendet, da sich die damit erzeugten Parser und Scanner sehr gut mit Scala kombinieren lassen. [Grei10]

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;

Ziele dieser Arbeit sind a) eine vollständige Grammatik des VHDL 2002 Standards für Coco/R[[2]](#footnote-2) zu entwickeln und darzulegen, wie die dabei auftretenden Probleme gelöst werden; b) die Grammatik so zu erweitern, dass ein Abstrakter Syntax Baum (AST) erstellt wird und c) den erstellten AST verwenden, um Eigenschaften des Source Code statisch zu überprüfen.

## Aufbau

Diese Bakkalaureatsarbeit ist wie folgt aufgebaut:

* 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 und wie dieser traversiert werden kann.
* Kapitel 4 erklärt anhand eines Beispiels, wie der erstellte AST verwendet werden kann, um Namenskonventionen zu überprüfen.

# LL(1) Konflikte und deren Beseitigung

„Eine Grammatik heißt LL(1) (d.h. analysierbar von links nach rechts mit linkskanonischen Ableitungen und 1 Vorgriffssymbol), wenn an jeder Stelle, an der man zwischen mehreren Alternativen wählen kann, gilt, dass die terminalen Anfänge dieser Alternativen paarweise disjunkt sind. Mit anderen Worten: der Parser muss jederzeit mit einem einzigen Vorgriffssymbol entscheiden können, welche von mehreren möglichen Alternativen er wählen soll.“ [Möss03]

Die im Standard [IEEE02] 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 erörtert, wie diese Konflikte gelöst werden können. Mit Hilfe dieser drei Varianten 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 die folgende Produktion

A = a b c | a b d.

in die 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 sehr gut 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 =

[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 =

Identifier

| stringLiteral

| SelectedName

| IndexedName

| SliceName

| AttributeName.

Prefix =

Name

| FunctionCall.

SliceName = Prefix lparen DiscreteRange rparen.

AttributeName = Prefix [Signature] ' Identifier [lparen Expression rparen].

Hier entsteht eine Linksrekursion, da SelectedName, IndexedName, SliceName und AttributeName mit dem rekursiven Nonterminal Prefix beginnen und Prefix wiederum ein Name sein kann, der z.B. wieder in einen SliceName abgeleitet werden kann.Durch die folgenden Umwandlung wird diese Rekursion vermieden:

Name = Prefix { NamePart }.

Prefix =

Identifier

| stringLiteral.

NamePart =

SelectedPart

| AttributePart

| SlicePart

| AssociationListPart.

SlicePart = lparen DiscreteRange rparen.

AttributePart = [Signature] ' Identifier [lparen Expression rparen].

Das nicht rekurisve Nonterminal Identifier und stringLiteral wurden zu einer neuen Produktion Prefix ohne Rekursion zusammengefasst. Name besteht jetzt aus Prefix und beliebig vielen NamePart wobei jeder Part nicht mehr mit Prefix beginnt, dadurch wurde die Rekursion von Prefix in eine Iteration von verschiedenen NamePart umgewandelt.

## Einsatz von Conflict Resolvers

Bei dem Einsatz von Conflict Resolvers kann 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

IdentifierList = Identifier { comma Identifier }.

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 nachstehenden 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 && next.kind!=\_EOF);

**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 dargelegt, 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 Typsystems 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 elegant 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:

**private** **static** <T> Option<T> toOption(T x) {**return** Option.apply(x);}

ReturnStatement<out ReturnStatement returnStmt > =

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

return [Expression<out expr>] semicolon

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

Die erzeugten optionalen 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 werden.

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 darin, dass die einzelnen Felder der AST-Klassen null sein können, was keine saubere und typsichere Lösung darstellt. Hier bietet auch Scala mit der Klasse Either[[5]](#footnote-5) eine Lösung an.

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

Target =

Name

| Aggregate.

Diese Möglichkeiten lassen sich durch folgende Klasse typsicher 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, dabei wird als Typ Scala Seq[[6]](#footnote-6) verwendet.

Beispiel:

IdentifierList<out Seq<Identifier> identifiers> =

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

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

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

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

## Mehrere Rückgabewerte

Um mehrere Werte zurück zugeben, werden die Scala Tuple*X* 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 Hierarchie

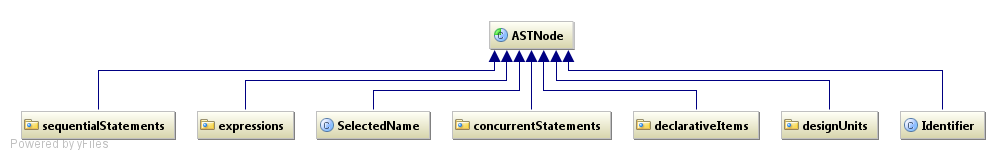


Abbildung 1: AST Klassenhierarchie Übersicht

Wie aus der Abbildung 1 ersichtlich, ist ASTNodedie Basisklasse aller AST-Knoten. Für fast alle Produktionen gibt es eine entsprechende Klasse, wobei die verschiedenen Knoten in den fünf Packages nach ihrer Art gruppiert sind. In diesen Packages gibt es wieder eine Basisklasse mit dem gleichen Namen, die von ASTNodeerbt und von der alle anderen Klassen im selben Package abgeleitet sind.

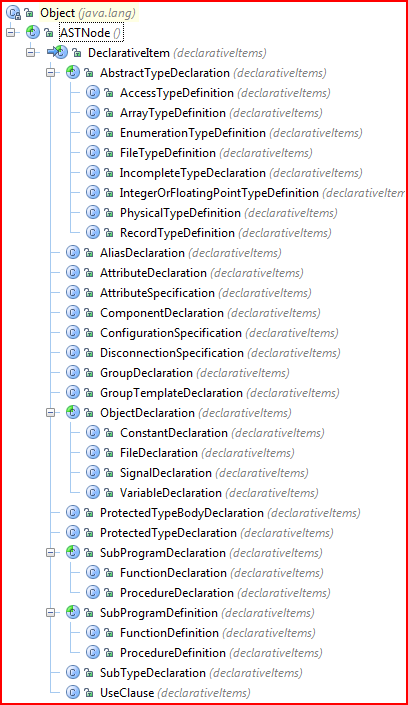
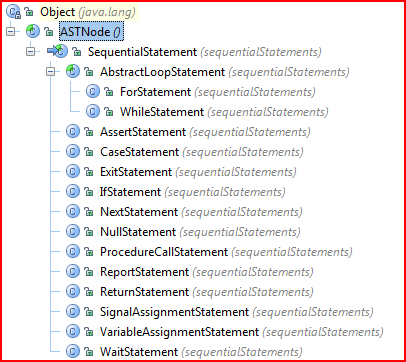


Abbildung 2: Klassenhierarchie für SequentialStatement Abbildung 3: Klassenhierarchie für DeclarativeItem

Mit dem in der Abbildung 2 und Abbildung 3 gezeigten Schema sind auch die Klassen in den Packages expressions, concurrentStatements und designUnitsimplementiert und bilden damit alle VHDL Konstrukte ab.

## AST Beispiele

Im folgenden Abschnitt wird anhand von konkreten Beispielen genauer verdeutlich, wie der konstruierte AST im Speicher dargestellt wird, wobei die einzelnen Kanten der Graphen die Variablen der verschiedenen Klassen darstellen. Es wird zuerst immer der entsprechende Programmcode gezeigt und anschließend der entsprechende Graph mit den AST-Knoten.

### WhileStatement

**while** expr **loop**

statement1;

statement2;

**end** **loop**;

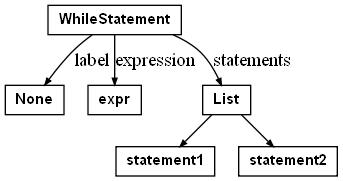


Abbildung 4: AST-Knoten für ein While-Statement

Ein While-Statement besteht aus einer Expression, die in der Variable expression gespeichert wird und einer Liste von Statements die in statements gespeichert werden.

### IfStatement

**if** expr1 **then**

statement1;

**elsif** expr2 **then**

statement2;

**else**

statment3;

**end** **if**;

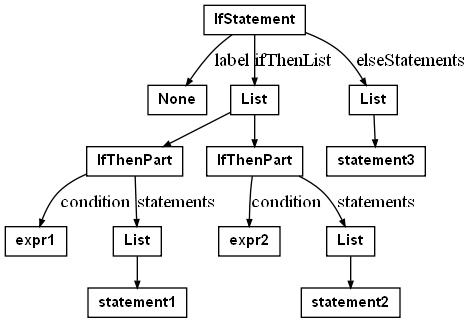


Abbildung 5: AST Knoten für ein If-Statement

Ein If-Statement besteht aus mehreren Zweigen, wobei die einzelnen Zweige Instanzen der Klasse IfThenPart sind, wo die Condition und die Liste der Statements gespeichert werden. Die Variable elseStatements enthält die Statements aus dem else-Zweig.

### LogicalExpression

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

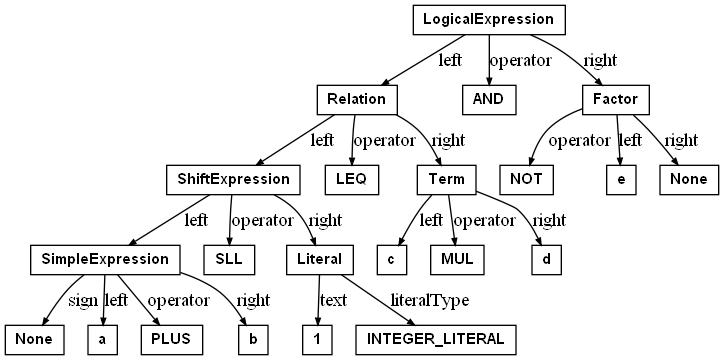


Abbildung 6: AST-Knoten für eine Logical-Expression

Eine Logical-Expression besteht wie alle Binary-Expression aus einer linken und rechten Expression und einem Operator der beide miteinader verknüpft.

### ConstantDeclaration

**constant** PI : real := 3.14159\_26535\_89793\_23846;

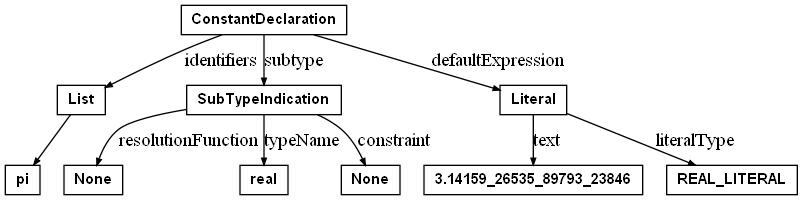


Abbildung 7: AST Knoten für eine Constant-Declaration

Die Deklaration für eine Konstante besteht aus einer Liste von Identifiers, einer Type Beschreibung und einem Default-Wert.

## AST Traversierung

Scala ermöglicht es duch sein elegantes Pattern-Matching System den erzeugten 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**

}

}

Die Funktion acceptList verwendet die Higher-order Funktion foreach, um auf jedes Element einer Liste wieder die Funktion acceptNode aufzurufen, um rekursiv alle Elemente zu besuchen.

# Beispiel-Anwendung

## Einleitung

In diesem Kaptitel wird erläutert, wie der erzeugte AST verwendet werden kann, um Namenskonventionen zu überprüfen, welche mit dem Java Werkzeug Checkstyle[[7]](#footnote-7) vergleichbar ist. Die Konfiguration steht dafür in einer JSON[[8]](#footnote-8)-Datei wobei es für jeden Knoten einen Eintrag mit einem regulären Ausdruck geben kann, der beschreibt wie die Identifier des Knotens genannt werden müssen.

Beispiel:

{

"NameChecker": {

"VariableDeclaration" : "^var[a-z][a-zA-Z0-9]\*$",

"ConstantDeclaration" : "^const[a-z][a-zA-Z0-9]\*$",

"SignalDeclaration" : "^sig[a-z][a-zA-Z0-9]\*$"

}

}

Diese Beispiel-Konfiguration besagt, dass Variablen mit dem Prefix „var“, Konstanten mit „const“ und Signale mit „sig“ beginnen müssen.

Angewandt auf folgenden Test-Code

**procedure** test **is**

**variable** varTest : integer;

**constant** x : real := 1.0;

**begin**

**end** test;

erzeugt das Programm die Fehlermeldung:

--test.vhd line 5 col 14: x does not match regex: ^const[a-z][a-zA-Z0-9]\*$

constant x : real := 3.0;

^

Die Anwendung gibt die Position und Zeile mit dem Identifier aus, der nicht dem Regulären Ausdruck entspricht.

## Implementierung

Die Implementierung ist sehr einfach, wenn man den in 3.6. beschriebenen Traversierungsmechanismus verwendet. Es wird bei jedem Knoten, der einen neuen Namen definiert, die check Funktion mit dem Identifier aufgerufen. Check vergleicht den Identifier mit dem in der Konfigurations-Datei angegebenen regulären Ausdruck. Wenn der Identifier diesem nicht entspricht, wird eine Fehler-Nachricht vom Typ CheckerMessage erzeugt und zu den vorhanden in checkerMessages hinzugefügt.

#### Source-Code

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

**def** acceptList[T <: ASTNode](list: Seq[T]) = list.foreach(acceptNode)

**def** check(identifier: Identifier): **Unit** = {

**val** regex = configuration.properties(node.getClass)

regex.findFirstMatchIn(identifier.text) **match** {

**case** None => checkerMessages += new CheckerMessage(identifier, regex)

**case** \_ =>

}

}

**def** checkList(identifiers: Seq[Identifier]) = identifiers.foreach(check)

node **match** {

**case** functionDefinition: FunctionDefinition =>

check(functionDefinition.identifier)

acceptList(functionDefinition.declarativeItems)

**case** procedureDefinition: ProcedureDefinition =>

check(procedureDefinition.identifier)

acceptList(procedureDefinition.declarativeItems)

**case** constDecl: ConstantDeclaration => checkList(constDecl.identifiers)

**case** varDecl: VariableDeclaration => checkList(varDecl.identifiers)

//… weitere AST Knoten

}

}

# 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 verschiedenen Strategien gelöst wurden. Weiters wurde dargelegt, wie diese Grammatik verwendet wurde, um einen AST erzeugen. Es wurde anhand von Code Beispielen gezeigt, wie dieser gespeichert wird. Anschließend wurde verdeutlicht, wie der AST traverstiert werden kann, um verschiedene Eigenschaften zu überprüfen. Im letzen Kapitel wurde der erzeugte AST verwendet, um Namenskonventionen zu überprüfen.

Aufbauend auf der geleisteten Arbeit ergeben sich viele interessante weiterführende Möglichkeiten um den erzeugten AST zu verwenden. Er könnte die Grundlage für einen Compiler, Interpreter oder für verschiedene Tools zur statischen Code Analyse, wie sie aus der Java Welt bekannt sind, bilden. Eine weitere Möglichkeit wäre, nach erfolgter semantischer Analyse, den Quellcode in eine andere Form umzuwandeln, z.B. in eine andere Hardwarebeschreibungssprache wie Verilog oder SystemC, oder in navigierbar HTML-Code.

# 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

[Grei10] Andreas Greilinger, Erweiterung des Eclipse-Plugins für Coco/R um Code-Vervollständigung in semantischen Aktionen

# Abbildungsverzeichnis

[Abbildung 1: AST Klassenhierarchie Übersicht 14](#_Toc286261248)

[Abbildung 2: Klassenhierarchie für SequentialStatement 14](#_Toc286261249)

[Abbildung 3: Klassenhierarchie für DeclarativeItem 14](#_Toc286261250)

[Abbildung 4: AST-Knoten für ein While-Statement 15](#_Toc286261251)

[Abbildung 5: AST Knoten für ein If-Statement 16](#_Toc286261252)

[Abbildung 6: AST-Knoten für eine Logical-Expression 16](#_Toc286261253)

[Abbildung 7: AST Knoten für eine Constant-Declaration 17](#_Toc286261254)

1. siehe www.ieee.org [↑](#footnote-ref-1)
2. siehe http://ssw.jku.at/Research/Projects/#Coco [↑](#footnote-ref-2)
3. siehe http://download.java.net/jdk7/docs/jdk/api/javac/tree/com/sun/source/tree/ReturnTree.html [↑](#footnote-ref-3)
4. siehe http://www.scala-lang.org/api/2.8.1/scala/Option.html [↑](#footnote-ref-4)
5. siehe http://www.scala-lang.org/api/2.8.1/scala/Either.html [↑](#footnote-ref-5)
6. siehe http://www.scala-lang.org/api/2.8.1/scala/collection/Seq.html [↑](#footnote-ref-6)
7. siehe http://checkstyle.sourceforge.net/config\_naming.html [↑](#footnote-ref-7)
8. siehe http://en.wikipedia.org/wiki/JSON [↑](#footnote-ref-8)