Ein Metamodell für abstrakte Syntaxbäume zum Einsatz in der Software-Modernisierung

Diplomarbeit im Fach Informatik

vorgelegt von

Joachim Lusiardi

29. September 2006

Bayerische Julius-Maximilians-Universität Würzburg

Lehrstuhl für Informatik II

Programmiersprachen und Programmiermethodik

betreut von

Prof. Dr. Jürgen Wolff von Gudenberg

Dipl. Inf. Gregor Fischer

Inhaltsverzeichnis

Einführung **vii**

I Theorie 1

1 Repräsentation von Quellcode **3**
1.1 Quellcode - Eine Folge von Zeichen 3
1.2 Quellcode - Eine Folge von Tokens 4
1.3 Quellcode - Ein Parsebaum 4
1.4 Quellcode - Ein abstrakter Syntaxbaum 5
1.5 Quellcode - Ein generischer abstrakter Syntaxbaum 6
1.6 Definition sprachspezifischer Fragmente 6
1.7 Definition des Begriffs Modell 6
1.8 Vorteile durch ein GAST Modell 7

2 Erstellung eines AST Modells **9**
2.1 Formalisiertes Vorgehen 9
2.1.1 Regeln zur Generierung von Attributen 10
2.1.2 Regeln zur Generierung von Klassen 10
2.2 Demonstration an der Sprache Lua 10
2.3 Abschließende Bemerkungen 14

3 Definition des GAST Modells **15**
3.1 Anmerkungen und Definitionen 15
3.2 Umfang des GASTs 15
3.3 Typbehandlung im GAST 17
3.4 Struktur eines GAST Dokuments 18
3.5 Deklaration externer Elemente 18
3.5.1 Deklaration von externen Basisdatentypen 19
3.5.2 Deklaration von Funktionen und Konstanten 20
3.5.3 Deklaration von externen, zusammengesetzten Datentypen 20
3.5.4 Nachteile dieses Vorgehens 21
3.6 Der Typ *Type* als Basistyp aller Typen 21

3.7 Referenzen auf Typen . . . 21
3.8 Basisdatenstrukturen . . . 22
3.8.1 GAST-Array . . . 23
3.8.2 GAST-List . . . 24
3.8.3 GAST-Map . . . 26
3.8.4 Zusammenfassung der Basisdatenstrukturen . . . 28
3.9 Anweisungen . . . 29
3.9.1 Block . . . 29
3.9.2 Deklaration von Variablen . . . 30
3.9.3 Deklaration von Funktionen . . . 31
3.9.4 Ausdrucksanweisung . . . 32
3.9.5 Zweifachauswahl . . . 33
3.9.6 Fallauswahl . . . 34
3.9.7 For-Schleife . . . 35
3.9.8 Foreach-Schleife . . . 36
3.9.9 Bedingungsgesteuerte Schleife . . . 37
3.9.10 Throw-Statement . . . 38
3.9.11 Monitored-Statement . . . 39
3.9.12 Return-Statement . . . 40
3.9.13 Break-Statement . . . 41
3.9.14 Continue-Statement . . . 42
3.10 Ausdrücke . . . 43
3.10.1 Literale . . . 43
3.10.2 Zugriffe auf Variablen . . . 44
3.10.3 Unäre Ausdrücke . . . 45
3.10.4 Binäre Ausdrücke . . . 46
3.10.5 Der Ternäre Ausdruck . . . 46
3.10.6 Funktionsdeklaration . . . 46
3.10.7 Funktionsaufruf . . . 47
3.11 Operatoren . . . 48
3.11.1 Logische Operatoren . . . 49
3.11.2 Relationale Operatoren . . . 49
3.11.3 Arithmetische Operatoren . . . 50
3.11.4 Sonstige Operatoren . . . 52
3.11.5 Weitere Operatoren . . . 53
3.12 Strukturierung des Quellcodes . . . 53
3.13 Erweiterung der Typauswahl . . . 54
3.13.1 Typdeklaration . . . 54
3.13.2 Felddeklarationen und Feldzugriffe . . . 55
3.13.3 Aufzählungsdeklaration . . . 57
3.13.4 Deklaration komplexer Typen . . . 59
3.13.5 Schnittstellendeklaration . . . 60
3.13.6 Klassendeklaration . . . 61

3.13.7 Erzeugen von Objekten 65
3.14 Sprachspezifische Fragmente 65

4 Ersetzung nicht unterstützter Elemente 67
4.1 Ersetzen der Inkrement- und Dekrementoperatoren 67
4.1.1 Äquivalenz von $++i$ und $(i = i + 1)$ 68
4.1.2 Äquivalenz von $i++$ und $((i = i + 1) - 1)$ 68
4.2 Umsetzen von *Und* und *Oder* 69
4.3 Ersetzen von *break* und *continue* mit Zielmarken 69
4.3.1 Verfahren für Break-Anweisungen 70
4.3.2 Verfahren für Continue-Anweisungen 71
4.4 Switch-Case Anweisungen 71
4.5 Klassische C Aufzählungen 72
4.6 Umsetzung von Casts in GAST 73

II Praxis 75

5 JaML - Java Markup Language 77
5.1 Einführung 77
5.2 Anforderungen an JaML 77
5.3 Implementierung als Plug-in für Eclipse 78
5.3.1 Relevante Informationen zu Eclipse 78
5.3.2 Details der Implementierung 79
5.3.3 Probleme bei der Implementierung 82
5.4 Die Implementierung der Kommandozeilenversion 85
5.5 Transformation von JaML nach Java 86
5.6 Test der Transformationen zwischen JAST und Java 86

6 JAST - Ein SAST für Java 87
6.1 Einführung 87
6.2 Unterschiede zwischen JaML und JAST 87
6.3 Transformation von JaML nach JAST 88
6.4 Transformation von JAST nach Java 89
6.4.1 Angabe der Qualifizierung der Typen 89
6.4.2 Wiedereinfügen der Klammern bei der Transformation nach Java 90
6.5 Testen der Transformation 91

7 GAST - Die Implementierung 93
7.1 Erstellung des GASTs 93
7.1.1 Schritt 1: Erstellen der PräGAST-Dokumente 93
7.1.2 Schritt 2: Erstellen des GAST-Dokuments 96

7.2 Transformation von GAST nach Java 99
7.2.1 Anwenden der XSL Transformation 99
7.2.2 Erstellen des Java Projektes 102

8 Handbuch für das Eclipse Plug-in 105
8.1 Features des Plug-ins 105
8.2 Anforderungen zur Verwendung des Plug-ins 105
8.2.1 Softwareanforderungen 105
8.2.2 Hardwareanforderungen 105
8.3 Installation des Plug-ins 106
8.3.1 Export des Plug-ins 106
8.3.2 Einbinden des Plug-ins in Eclipse 106
8.4 Einsatz des Plug-ins 107
8.4.1 In Eclipse 107
8.4.2 Auf der Kommandozeile 107

9 Ergebnis und Diskussion 111
9.1 Implementierung von JaML 111
9.2 Implementierung von JAST 111
9.3 Die Definition des GASTs 112
9.4 Die Implementierung $Java \leftrightarrow GAST$ 112
9.4.1 Nicht in Java umgesetzte GAST-Elemente 112
9.4.2 Nicht in GAST umgesetzte Java Konstrukte 113
9.5 Übersicht über Transformationen 114
9.6 Vergleich der Formate 114
9.6.1 Betrachtung des HelloWorld-Projekts 115
9.6.2 Betrachtung des CodeCollection-Projekts 115
9.7 Größenreduzierung der `GAST`-Dateien 117
9.8 Ausblick 117

Einführung

Ziel dieser Diplomarbeit ist die Entwicklung einer sprachunabhängigen Darstellung von Quellcode, die dann in verschiedensten Bereichen flexibel einsetzbar sein soll.

In Unternehmen sammeln sich im Lauf der Zeit eine Vielzahl von spezifischen, auf das Unternehmen zugeschnittene, Softwarelösungen an. Ergeben sich nun tiefgreifende Umstellungen der IT-Infrastruktur, sei es durch einen Wechsel des Betriebssystems oder neue Anforderungen an die Funktionalität der Software, so stellt sich oft folgendes Problem:

Wie kann man die „Altlasten“ in das neue System integrieren und übernehmen?

Eine sprachunabhängige Darstellung eröffnet hier nun eine Reihe von Möglichkeiten. Zum einen kann Quellcode sprachübergreifend analysiert und mit Metadaten versehen werden. Dadurch ist es möglich, so genannte Softwaremodernisierungswerkzeuge mit den Werkzeugen der Entwickler besser zu kombinieren.

Weiterhin kann man auf Basis eines GAST Modells und mit Hilfe der MDA[22] eine automatische Übersetzung auf Quellcodeebene erreichen.

Teil I

Theorie

Kapitel 1

Repräsentationsformen von Programmcode

In ersten Teil der Arbeit wird nun der theoretische Aspekt der Entwicklung eines Metamodells behandelt. Dies umfasst einen kurzen Exkurs über die Darstellung von Quellcode, einer Einführung in die Erstellung von AST Modellen und schließlich die Definition des GAST Modells.

Für die Repräsentation von Quellcode verschiedener Programmiersprachen existieren zahlreiche Möglichkeiten. Die Wahl der Darstellungsform ist natürlich abhängig von der zu lösendenden Aufgabe und oft werden verschiedene Formen parallel benötigt.

1.1 Quellcode - Eine Folge von Zeichen

Die Darstellung des Quellcodes als Folge von Zeichen ist die gewöhnlich genutzte Form, um als Programmierer Quellcode zu bearbeiten.

```
1   i = i + 1;
```

Listing 1.1: eine einfache Anweisung

Diese Darstellung vermag etliche Dinge nicht zu leisten, da keine weiteren Informationen außer dem reinen Quellcode vorhanden sind. Zum einen besteht keine Möglichkeit, den Programmierer durch optische Markierungen im Code zu unterstützen, zum anderen sind nur sehr eingeschränkte Möglichkeiten zur Analyse vorhanden.

Mehr Informationen werden erst durch Darstellung als Tokenstrom oder als Baumstruktur zugänglich. Diese Möglichkeiten heißt es, im folgenden zu beschreiben.

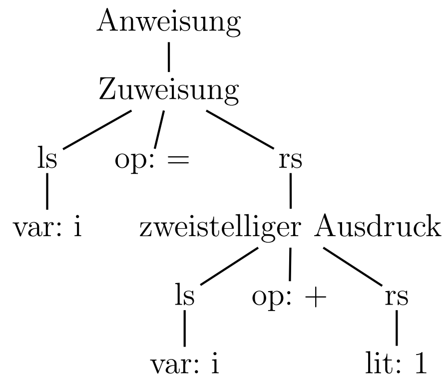


Abbildung 1.1: Ein möglicher Parsebaum für `i = i + 1`

1.2 Quellcode - Eine Folge von Tokens

Die Anweisung aus Listing 1.1 kann durch einen Lexer beispielsweise in folgenden Tokenstrom zerlegt werden:
[id: i] [ws] [op: =] [ws] [id: i] [ws] [op: +] [ws] [lit: 1] [sc]
Dadurch erhält man mehr Informationen über den Quellcode. Zum einen werden überflüssige Zeichen (z.B. Whitespaces) markiert, zum anderen Operatoren, Identifier und Literale identifiziert. Eventuell vorhandene Schlüsselwörter werden ebenfalls gekennzeichnet.

Diese Informationen sind zum Beispiel ausreichend, um in einem Editor eine einfache optische Hervorhebung von Syntaxelementen der Sprache zu ermöglichen.

1.3 Quellcode - Ein Parsebaum

Die Anweisung aus Listing 1.1, repräsentiert durch den Tokenstrom aus Kapitel 1.2, wird durch einen Parser beispielhaft in den Parsebaum in Abbildung 1.1 umgewandelt.

Der Parser gewinnt wieder mehr Informationen über das Programm. Einzelne Tokens werden zu syntaktischen Gruppen zusammengefasst. Beispielsweise erkennt der Parser im obigen Beispiel eine Anweisung, die eine Zuweisung eines binären Ausdrucks an eine Variable ist. Ebenso erkennt der Parser Ausdrücke und hängt diese entsprechend der Operatorhierarchie und der Assoziativität in den Parsebaum ein.

Allerdings sind in dieser Form des Syntaxbaums noch überflüssige und redundante Elemente enthalten. Überflüssige Elemente sind in unserem Beispiel die Knoten für linke und rechte Seite (rs / ls). Die Redundanz ergibt sich aus Vorgaben der Sprache.

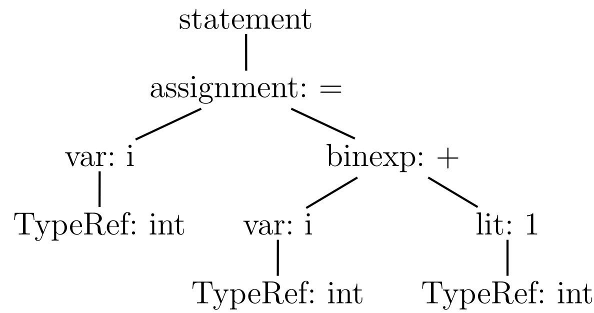


Abbildung 1.2: Ein abstrakter Syntaxbaum

Diese Darstellungsform wird zum Beispiel verwendet, um Hervorhebungen der Syntax zu erhalten und gleichzeitig Anfragen zum Code beantworten zu können. Beispielsweise sind im JDT Eclipse Plug-in Anfragen über den Ort einer Variablendeklaration oder automatische Hinweise zur Signatur einer Methode möglich.

1.4 Quellcode - Ein abstrakter Syntaxbaum

In einem weiteren Abstraktionschritt werden die Parsebäume zu abstrakten Syntaxbäumen abstrahiert.

Der Unterschied zwischen einem Parsebaum und einem abstrakten, attributierten Syntaxbaum besteht nach den Drachenbüchern ([8] und [9]) in einer Abstraktion der Struktur der Bäume. Ein abstrakter Syntaxbaum ist eine verdichtete Form des Parsebaums und ermöglicht es, Sprachkonstrukte als solche zu erkennen und darzustellen.

Werden weitere Informationen, die die Semantik betreffen, hinzugefügt, so spricht man von attributierten Syntaxbäumen.

Aus unserem Beispielbaum aus Abbildung 1.1 wird so der abstrakte Syntaxbaum aus Abbildung 1.2.

Der abstrakte Syntaxbaum ermöglicht, ebenso wie der Parsebaum, detaillierte Anfragen und Untersuchungen des Programmcodes. Als weiterer Vorteil kann gelten, dass leicht reproduzierbare Elemente des konkreten Syntaxbaums entfernt wurden und schwer zu erhaltende Informationen hinzugefügt werden.

Bezieht sich ein abstrakter Syntaxbaum auf eine spezielle Programmiersprache, so wird im Folgenden von einem specific abstract syntax tree (**SAST**) gesprochen.

1.5 Quellcode - Ein generischer abstrakter Syntaxbaum

Der generische abstrakte Syntaxbaum (**GAST**, generic abstract syntax tree) ist eine weitere Stufe der Abstraktion bei der Darstellung von Quellcode. Dabei wird versucht, alle Sprachspezifika durch allgemeine Konstrukte gleicher Semantik zu ersetzen. Dies begünstigt Transformationen zwischen dem Format des GASTs und real existierenden Sprachen bzw. SASTs.

Leider ist es prinzipiell unmöglich, für alle Sprachen alle Konstrukte und Schlüsselwörter in einen GAST aufzunehmen, da dies den GAST zu sehr aufblähen würde (siehe auch Abschnitt 3.2). Beispielsweise bietet Java Schlüsselwörter wie *strictfp* oder *volatile* an, die keine Entsprechung im GAST finden. Diese werden dann durch **sprachspezifische Fragmente** ausgedrückt. Eine Definition für ein sprachspezifisches Artefakt findet man im nächsten Abschnitt, das genaue Format wird jedoch erst in Kapitel 3.14 definiert.

1.6 Definition sprachspezifischer Fragmente

Ein sprachspezifisches Fragment ist ein Teil eines GASTs und repräsentiert ein Element einer Quellsprache, welches nicht direkt im GAST darstellbar ist, trotzdem aber für die Semantik des Programms wichtig ist. Dies können spezielle Modifier sein, die nicht im GAST definiert sind, oder auch Teilbäume des behandelten SASTs.

Die folgenden Informationen müssen in einen sprachspezifischen Fragment enthalten sein. Zum einen die Quellsprache, aus der das Fragment stammt, da nur so nachträgliche Transformationen anwendbar sind. Zum anderen der Teil des SASTs, der nicht im GAST darstellbar war.

Bei Transformationen des GASTs in den entsprechenden SAST muss dann jeweils das sprachspezifische Fragment umgesetzt werden. Dies kann per Hand, oder besser, durch entsprechende Bibliotheken durchgeführt werden.

Diese Umsetzungen wurden im Verlauf der Arbeit nur zwischen Java als Quell- und Java als Zielsprache umgesetzt, das dies sonst den Rahmen der Arbeit gesprengt hätte.

1.7 Definition des Begriffs Modell

Nach [24] ist ein Modell ein Abbild einer realen oder abstrakten Sache, welches in einem oder mehreren Aspekten in einer Abstraktionsbeziehung zum betrachteten Objekt steht.

Auch [7] definiert in Abschnitt 2.2.2 den Begriff des Modells. Hier ist ein Modell eines Systems eine Spezifikation des Systems und seiner Umgebung unter

einem oder mehreren Aspekten. Die Darstellung dieser Spezifikation erfolgt in Wort und Bild, wobei die Sprachen formal oder informal sein dürfen.

Ein wichtiges Ziel der Modellierung ist immer eine Reduktion durch Entfernen unwichtiger, redundanter Details und das Fixieren auf relevante Aspekte. Der Grad der Formalisierung, sowohl der Syntax als auch der Semantik, ist ein weiterer wichtiger Aspekt der Modellierung.

1.8 Vorteile durch ein GAST Modell

Die Transformation zwischen verschiedenen Programmiersprachen ist eine Hauptmotivation für die Einführung eines GAST Modells. Das Problem ist hierbei nicht die Transformation einzelner Sprachen in andere, also eine Beziehung 1 zu 1, sondern eine universelle Möglichkeit n Quellsprachen in m Zielsprachen zu transformieren.

Der Aufwand zur Lösung liegt hierbei in der Größenordung von $n * m$, also bei ungefähr quadratischem Aufwand.

Durch eine geschickte Definition des GASTs kann dieser Aufwand nun reduziert werden. Besteht für jede der n Quellsprachen die Möglichkeit der Transformation in den GAST und für jede der m Zielsprachen die Möglichkeit, diese aus dem GAST zu erzeugen, so benötigt man nur noch $n + m$ Transformationen. Leider ist dieser Aufwand nur eine untere Schranke, da die sprachspezifischen Fragmente weiterhin direkt zwischen den Sprachen übersetzt werden müssen. Der Umfang der Sprachspezifischen Fragmente ist im Vergleich zum Umfang der kompletten Sprache meist stark reduziert. Daher sinkt der Aufwand von $n*m$ auf $n+m+R(n, m)$ mit $R(n, m)$ als Repräsentation des Aufwands zur Transformation der sprachspezifischen Fragmente.

Kapitel 2

Erstellen eines Modells für einen abstrakten Syntaxbaum

Gute Modelle für abstrakte Syntaxbäume real existierender Programmiersprachen stellen wichtige Grundlagen dar. Sie dienen als Basis für die Definition des GAST- Modells und auf ihnen basieren auch die Transformationen zwischen den einzelnen SAST Modellen und dem GAST.

Ebenfalls ist ein solches Modell Grundlage eines jeden Compilers und daher auch in der Praxis von hoher Bedeutung.

2.1 Formalisiertes Vorgehen

Nun soll ein formalisiertes Verfahren beschrieben werden, mit dem aus einer Backus-Naur Form [29] ein Modell für einen spezifischen abstrakten Syntaxbaum (SAST) in Form eines UML Klassendiagramms gewonnen werden kann. Da neben den Informationen aus der Backus-Naur Form, also der Syntax der Sprache, auch stets die Semantik der Sprache betrachtet werden muss, kann dieses Verfahren nicht zu einem vollautomatischen Algorithmus ausgebaut werden. Deshalb, und auch weil die Regeln nur am Rande dieser Diplomarbeit erarbeitet wurden, sollten sie nur als Grundlage für die Erstellung des Modells und als Richtlinien für weitere Entwicklungen dienen.

Die Eingabe ist eine Backus-Naur Form (BNF) mit Nichtterminalen $\mathbf{N}$ und Terminalen $\mathbf{T}$. Die nun folgenden Regeln beschreiben eine Art Normalform für BNF, die leichter in ein AST Modell umgesetzt werden kann, in dem bereits Klassen und Attribute identifiziert werden. Auch Assoziationen und Generalisierungen können durch das Verfahren gewonnen werden.

Für diesen Abschnitt definieren wir: $n, n_1, n_2, \dots \in \mathbf{N}$ seien Nichtterminale, $t, t_1, t_2, \dots \in \mathbf{T}$ seien Terminale und $s, s_1, s_2, \dots \in \{\mathbf{T} \cup \mathbf{N}\}^+$ seien Sätze aus Terminalen und Nichtterminalen.

2.1.1 Regeln zur Generierung von Attributen

Zunächst werden zwei Regeln beschrieben, die es erlauben, Attribute der späteren Klassen zu identifizieren. Die Typen der Attribute werden ebenfalls festgelegt.

Umformung 1: Zum ersten werden alle Regeln der Form $n \rightarrow t_1 \mid t_2 \mid \cdots$ bearbeitet. Ein solches Nichtterminal n wird als Attribut behandelt. Dazu wird eine Aufzählung (in der UML eine Enumeration) definiert, deren Werte $t_1, t_2, \cdots$ die Terminale auf der rechten Seite der Regel darstellen. Alle Vorkommen des Nichtterminals n in den weiteren Regeln der BNF markiert man als Attribut und verwendet sie später entsprechend in den Klassen, die definiert werden.

Umformung 2: Nichtterminale, die in der Backus-Naur Form wie Terminale verwendet werden, also nicht durch weitere Regeln erklärt sind, müssen abhängig vom Kontext und unter Berücksichtigung der Semantik behandelt werden. Für Attribute bietet sich in diesem Fall als Datentyp **String** an.

2.1.2 Regeln zur Generierung von Klassen

Die drei nun aufgeführten Regeln dienen der Identifikation von Klassen, deren Attributen und ihren Beziehungen zu anderen Klassen.

Umformung 3: Als nächstes werden Regeln der Form $n \rightarrow s_1 \mid s_2 \mid \cdots$ behandelt. Die Sätze $s_1, s_2, \cdots$ werden in zusätzliche Regeln der Form $n_i \rightarrow s_i$ umgewandelt. Ausgenommen sind dabei Sätze, die nur aus einem weiteren Zeichen $s \in \mathbf{N} \cup \mathbf{T}$ bestehen. Die Bezeichnung des neuen Nichtterminals n_i sollte dabei die Bedeutung des Satzes s_i wiederspiegeln. Dies ist jedoch selten automatisch möglich. Das Nichtterminal n wird zu einer abstrakten Klasse umgeformt. Die Alternativen auf der rechten Seite werden zu Klassen, die von der entstandenen abstrakten Klasse erben. Ihre Attribute und Assoziationen werden durch die rechten Seiten der Regeln $n_i \rightarrow s_i$ erklärt.

Umformung 4: Nichtterminale, die in der Backus-Naur Form nicht durch weitere Regeln erklärt werden, wurden bereits im Abschnitt 2.1.1 über die Generierung von Attributen behandelt. Der Kontext und die Semantik der Sprache können auch eine Abbildung des Nichtterminals als Klasse fordern.

Umformung 5: Regeln ohne Alternativen mit gemischter rechter Seite werden, unter Beachtung der Semantik der betrachteten Sprache, von den Nichtterminalen befreit. Diese werden, da nur der Syntax dienlich, einfach gelöscht. Dadurch erhält man die Attribute und Assoziationen der Klasse.

2.2 Demonstration an der Sprache Lua

Lua ist eine prozedurale Sprache, deren Einsatzmöglichkeiten von eigenständigen Programmen bis hin zum Einsatz als programm-interne Skriptsprache reichen. Neben der prozeduralen Ausrichtung der Sprache gibt es Möglichkeiten, Lua funktional oder Objekt-basiert zu verwenden.

Die Möglichkeit, Lua als interne Skriptsprache zu verwenden, wird unter anderem in kommerziellen Programmen wie z.B. World of Warcraft, Adobe Lightroom und anderen [30] genutzt.

Folgende Zeilen sind ein Ausschnitt aus der Syntax der Sprache Lua (siehe [17]).

$$\begin{aligned}
binaryoperator &\rightarrow \textbf{`+'} \mid \textbf{`-'} \mid \cdots \mid \textbf{`and'} \mid \textbf{`or'} && (2.1)\\
expression &\rightarrow \textbf{`nil'} \mid \cdots \mid number \mid \cdots \\
&\quad \mid expression\ binaryoperator\ expression && (2.2)\\
statement &\rightarrow \textbf{`local'}\ \textbf{`function'}\ name\ funcbody \\
&\quad \mid \textbf{`repeat'}\ block\ \textbf{`until'}\ expression \\
&\quad \mid \cdots && (2.3)\\
namelist &\rightarrow Number\ \textbf{`:'}\ Name\ (\ \textbf{`,'}\ Name\ )^* && (2.4)
\end{aligned}$$

Betrachtet man nun Regel 2.1, so wendet man darauf Umformung 1 an. Als Ergebnis erhält man dadurch ein Attribut *binaryoperator*. Dessen Typ wird durch die Enumeration *binaryoperatorType* aus Abbildung 2.1 gegeben.

<<enumeration>> binaryoperatorType
plus minus ... and or

Abbildung 2.1: Enumeration für den Typ der binären Operatoren

Betrachtet man nun Regel 2.2, so kann man Umformung 3 anwenden und erhält die folgenden neuen Regeln:

$$\begin{aligned}
expression &\rightarrow \textbf{`nil'} \mid \cdots \\
&\quad \mid number \mid \cdots \\
&\quad \mid binaryExpression && (2.5)\\
binaryExpression &\rightarrow expression\ binaryoperator_{attr}\ expression && (2.6)
\end{aligned}$$

Anschließend wird Regel 2.2 entfernt.

Das Nichtterminal *expression* wird später im Modell zu einer abstrakten Klasse. Die Sätze auf der rechten Seite werden zu eigenständigen Klassen, die von der abstrakten Klasse *expression* erben. Das Klassendiagramm aus Abbildung 2.2 stellt diesen Sachverhalt da.

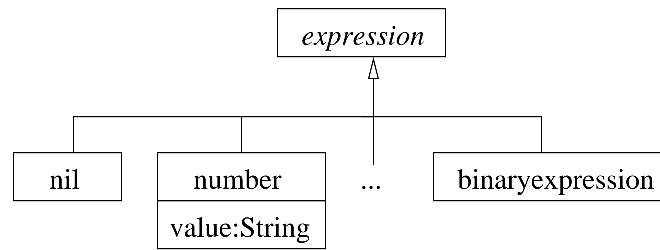


Abbildung 2.2: Klassendiagramm zum Element *expression*

Das Nichtterminal *number* wurde hierbei nicht weiter spezifiziert, deshalb muss seine Semantik herangezogen werden, um weitere Entscheidungen treffen zu können. Da das Nichtterminal numerische Konstanten repräsentiert, wird der entstehenden Klasse ein Attribut *value* hinzugefügt. Der Typ dieses Attributs wird in diesem Fall auf ein sehr weitgefasstes **String** gesetzt. Dies ist eine Mischung zwischen Umformung 2 und Umformung 4 und beruht auf Versuchen und Erfahrung.

Das Terminal **'nil'** wird zu einer weiteren Klasse.

Die Klasse *binaryExpression* wird aus Regel 2.6 direkt gewonnen. Das Ergebnis sieht man in Abbildung 2.3.

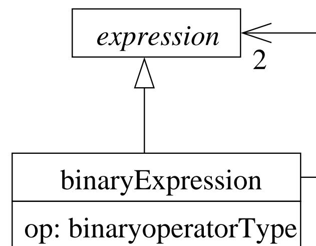


Abbildung 2.3: Klassendiagramm zum Element *binaryExpression*

Auf Regel 2.3 wird zunächst Umformung 3 angewandt und es werden folgende neue Regeln eingeführt:

$$statement \rightarrow localFunctionDeclaration \mid repeatUntilLoop \mid \cdots \quad (2.7)$$

$$localFunctionDeclaration \rightarrow \textbf{`local'}\ \textbf{`function'}\ name\ funcbody \quad (2.8)$$

$$repeatUntilLoop \rightarrow \textbf{`repeat'}\ block\ \textbf{`until'}\ expression \quad (2.9)$$

Daraus ergibt sich direkt die abstrakte Klasse *statement* und die Klassen *localFunctionDeclaration* und *repeatUntilLoop*. Das zugehörige Klassendiagramm

findet man in Abbildung 2.4. Anschließend wird Regel 2.3 aus der Grammatik entfernt.

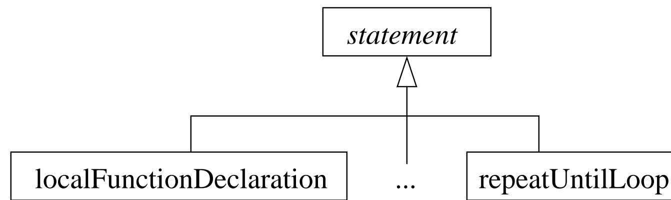


Abbildung 2.4: Klassendiagramm zum Element *statement*

Auf die neuen Regeln 2.8 und 2.9 wird dann Umformung 5 angewendet und man erhält die Diagramme für die beiden Klassen *localFunctionDeclaration* und *repeatUntilLoop* in den Abbildungen 2.5 und 2.6.

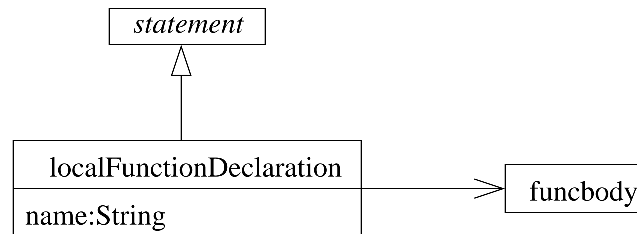


Abbildung 2.5: Klassendiagramm zum Element *localFunctionDeclaration*

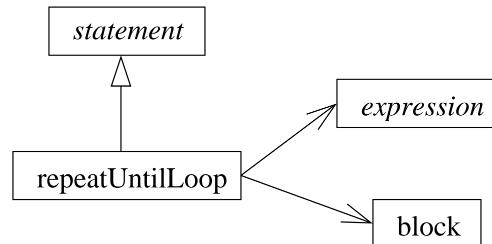


Abbildung 2.6: Klassendiagramm zum Element *repeatUntilLoop*

Als explizites Beispiel für den Konflikt zwischen den Umformungen 2 und 4 dient Regel 2.4. Diese Produktion ist nicht in der Lua Syntax enthalten und dient hier nur zur Verdeutlichung. In dieser speziellen Produktion sind sowohl das Nichtterminal $Number$ als auch das Nichtterminal $Name$: nicht weiter erläutert. Die Wiederholung der Namen wird in einem Klassendiagramm am besten durch eine Assoziation zur Klassen *Namen* mit Multiplizität $1..*$ dargestellt werden.

Für die Umsetzung des Nichtterminals *Number* bestehen nun zwei Möglichkeiten, ersten die Modellierung als Klasse, zweitens die Modellierung als Attribut an der Klasse *namelist*. Abbildung 2.7 zeigt beide Alternativen der Modellierung.

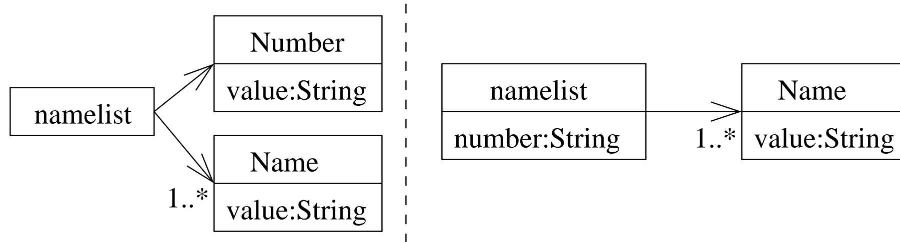


Abbildung 2.7: Klassendiagramm zum Konflikt der *namelist*

2.3 Abschließende Bemerkungen

Wie bereits betont, kann dieses Kapitel keinen vollständigen Algorithmus bieten. Dies wäre über die Grenzen dieser Diplomarbeit gegangen. Die Umformungen, die im ersten Abschnitt angegeben wurden, sind weder erschöpfend, noch eindeutig in ihrer Anwendung. Sie bieten aber eine Anleitung, die die Modellierung des AST Modell vereinfacht. Das Erstellen eines Modells für einen abstrakten Syntaxbaum einer Sprache benötigt immer noch profunde Kenntnisse der behandelten Sprache, Intuition, Erfahrung und Zeit.

Umformungen, die alle Arten von Wiederholungen in der Backus-Naur Form betreffen, werden bisher durch Umformungsregeln überhaupt nicht betrachtet. Dies kann in der Zukunft durch weitere Untersuchungen jedoch eventuell noch ergänzt werden.

Kapitel 3

GAST - Der generische abstrakte Syntaxbaum

3.1 Anmerkungen und Definitionen

In den folgenden Abschnitten werden GAST-Ausschnitte in Codeform notiert. Dies dient lediglich der Übersichtlichkeit im Vergleich zur XML-Repräsentation. Dabei werden durch // einzeilige Kommentare eingeleitet, die im GAST nicht repräsentiert werden. Ein Parser, der GAST Code parsen und in das entsprechende XML Dokument umsetzt, wurde nicht entwickelt.

In den nachstehenden Kapiteln werden Sprachen bezüglich ihres Typsystems in zwei Kategorien eingeteilt:

Getypte Sprachen sind Sprachen, deren Variablen, Attribute bei Klassen, Konstanten, etc. nur mit Angabe eines Typs deklariert werden können. Diese Typen sind dann fest.

Ungetypte Sprachen sind Sprachen, deren Variablen, Attribute bei Klassen, Konstanten, etc. ohne Angabe eines Typs deklariert werden können. Beispielsweise wird der Typ einer Variablen erst durch Zuweisungen von Werten dynamisch auf den Typ des Wertes festgelegt.

3.2 Umfang des GASTs

Eine Überlegung bei der Definition des GASTs ist der Umfang. Einerseits kann man das Gewicht auf eine schlanke Definition legen und möglichst wenige Konstrukte erlauben, bzw. manche Konstrukte einschränken (z.B. einfache und mehrfache Vererbung). Andererseits kann man das Gewicht auch auf eine große Universalität legen und nahezu alle bekannten Elemente in den GAST aufnehmen. Dies hat dann allerdings zur Folge, dass die Definition des GASTs schnell unübersichtlich wird.

Allerdings ist auch die universellste Definition des GASTs endlich und kann nicht um beliebige Element beliebiger Sprachen erweitert werden.

Eine wichtige Tatsache ist jedoch folgendes: Es genügt, die betrachteten Sprachen auf ihren Sprachkern zu reduzieren und nur diese reduzierte Form zu betrachten. Beispielsweise existieren in Java-Schreibweisen für die gleichzeitige Deklaration von Variablen.

```
//mehrfache Deklaration
int b=1,c=0;

//alternative Darstellung
int b=1;
int c=0;
```

Listing 3.1: mehrfache Deklarationen in Java

Die Sprache Lua besitzt ebenfalls viele solcher syntaktischen Alternativen, z.B. ist der Zugriff auf Arrayeinträge in mehreren Schreibweisen möglich.

```
array = { name = noob, ... };

//Zugriff durch Punkte
nick = array.name;

//traditioneller Zugriff
nick = array["name"];
```

Listing 3.2: Arrayzugriffe in Lua

Man kann nun daraus resultierend wie folgt argumentieren: Unterstützt der GAST ein Konstrukt einer Quellsprache, so reduzieren sich mögliche Probleme auf Transformationen in die diversen Zielsprachen, die das betrachtete Konstrukt nicht unterstützen. Siehe dazu Tabelle 3.2. In dieser Tabelle bedeutet ein 'ok', dass

Unterstüzung im GAST	Transformation in GAST	Transformation in Zielsprache	
		Zielsprache ja	Zielsprache nein
Ja	ok	ok	?
Nein	?	ok	

Tabelle 3.1: Analyse zum GAST Umfang

keine Probleme auftreten können, ein '?' bedeutet, dass Probleme dann auftreten, wenn das Programm das betrachtete Konstrukt verwendet. Existiert ein Element im GAST, aber nicht in der Quellsprache treten bei der Transformation in den GAST keine Probleme auf.

Dieser Problematik tritt diese Arbeit durch die Definition der sprachspezifischen Fragmente im GAST entgegen. In Ausnahmefällen kann somit das entsprechende Fragment des SASTs doch in den GAST abgebildet werden.

Transformationen von GAST-Instanzen in SAST-Instanzen können jedoch nur dann automatisch semantisch äquivalent umgesetzt werden, wenn keinerlei sprachspezifische Fragmente im GAST enthalten sind. Ansonsten sind zusätzliche Bearbeitungsschritte durch weitere Programme oder per Hand nötig. Während dieser Schritte werden dann die sprachspezifischen Fragmente der ursprünglichen Sprache entfernt und durch Elemente der Zielsprache ersetzt.

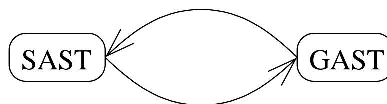
3.3 Überlegungen zur Behandlung von Typen im GAST

Zuerst war angedacht, wie bei einem Syntaxbaum, auf Typen lediglich durch sprachspezifische Fragmente zu verweisen. Der GAST wäre auf diese Art frei von jeglichen Typen gewesen und nur syntaktische Konstruktionen wären in die Definition aufgenommen worden.

Diese Einschränkung der Sprachunabhängigkeit in Form der sprachabhängigen Typen ist natürlich sehr gravierend. Trotzdem wäre diese Variante für viele Einsatzbereiche ausreichend leistungsfähig. Exemplarisch wären dies:

- Aufbau von Datenflussgraphen und Analyse auf sprachunabhängiger Ebene
- Aufbau und Analyse von Kontrollflussgraphen
- sprachunabhängige Anwendung von Metriken
- Umsetzung in UML über XMI und ebenso zurück, ein roundtrip-engineering ist denkbar

Stattdessen wird folgender Ansatz favorisiert und zur Grundlage dieser Arbeit. Neben einigen Basistypen und Datenstrukturen werden die verwendeten Typen in einer Liste der externen Typen gespeichert und im Programm wird nur auf Einträge dieser Liste verwiesen. Dadurch sind Transformationen zwischen einer Sprache (als Ziel und Quelle) und dem GAST immer als roundtrip möglich.



3.4 Struktur eines GAST Dokuments

Ein Instanzdokument des GAST-Modells wird in zwei Bereiche gegliedert (siehe Abbildung 3.1):

- den Programmbereich, der den Inhalt des dargestellten Programms repräsentiert. Hier werden alle Informationen abgelegt, die aus dem behandelten Projekt stammen. Diesen Bereich repräsentiert das Element *Program*. Weitere Erklärungen hierzu findet man in Abschnitt 3.12.

- den Bereich für externe Deklarationen. Hier werden Typen, Konstanten und Funktionen deklariert, die im Programmbereich verwendet, aber nicht zum Programmcode des Projekts gehören. Das sind die Basisdatentypen der Quellsprache und entsprechende Typen, Konstanten und Funktionen aus den verwendeten Bibliotheken der Sprache. Diesen Bereich repräsentiert im GAST das Element *ExternDeclarationList*. Eine genauere Definition erfolgt in Abschnitt 3.5.

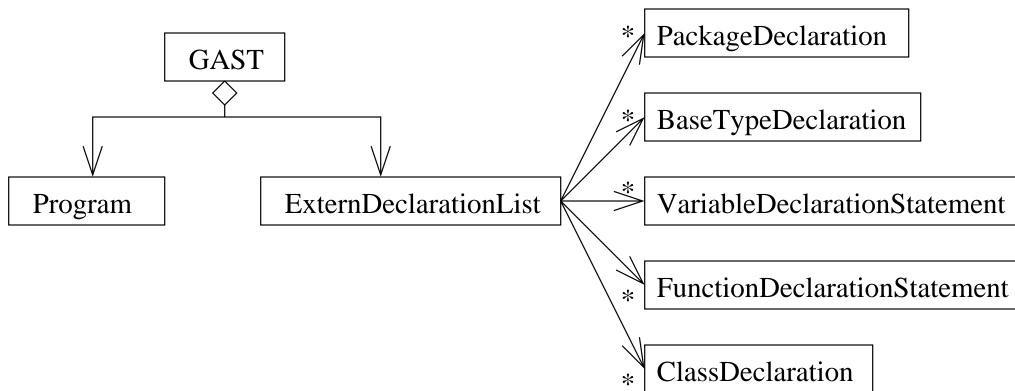


Abbildung 3.1: Struktur eines GAST-Dokuments

3.5 Deklaration externer Elemente

Externe Elemente werden im GAST Modell im Element *ExternDeclarationList* abgelegt. Dies umfasst die Deklaration von prozeduralen Elementen wie Basistypen (repräsentiert durch *BaseTypeDeclaration*-Elemente), Funktionen (repräsentiert durch *FunctionDeclaration*-Elemente) und Variablen oder Konstanten (repräsentiert durch *VariableDeclaration*-Elemente). Weiterhin können Elemente objektorientierter Sprachen wie Paketdeklarationen und Klassen eingebunden werden.

3.5.1 Deklaration von externen Basisdatentypen

Basisdatentypen sind in vielen Sprachen vorhandene primitive Typen, wie z.B. aus Java char, byte, short, double und viele weitere ([15] Kapitel 4). Bei den Basisdatentypen wird deshalb versucht, nur allgemeine, sprachunabhängige Eigenschaften der Typen zu speichern. Die entsprechenden Element des GAST Modells zu ihrer Deklaration wird in Abbildung 3.2 als Klassendiagramm eingeführt.

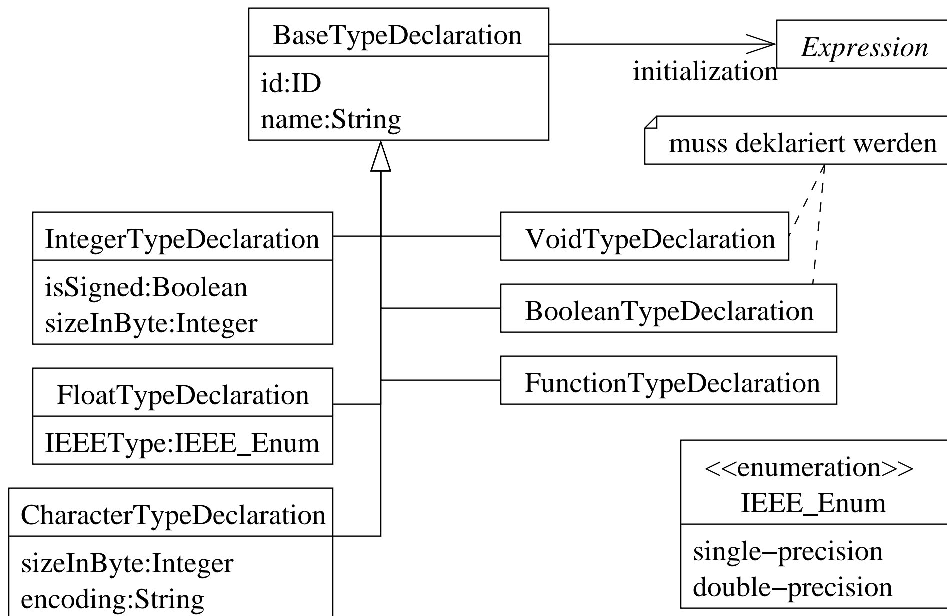


Abbildung 3.2: Externe Basistypen im GAST

Die einzelnen Elemente werden nun erklärt. Allen Elementen gemein sind die Attribute *id* und *name*. Die ID dient zur Identifizierung des Typs im Programmbereich des GASTs, der Name ist der in der Quellsprache verwendete Bezeichner und wird für die Transformationen des GASTs in die entsprechende Sprache benötigt. Die als *initialization* angebundene *Expression* dient als Initialisierungswert bei Standardinitialisierungen, wie sie der Konstruktor bei den Basisdatenstrukturen *Array* (siehe Abschnitt 3.6) und *List* (siehe Abschnitt 3.7) vornimmt.

Ein *IntegerTypeDeclaration*-Element besitzt als Attribute zusätzlich *isSigned* und *sizeInByte*. Das boolesche Attribut *isSigned* gibt an, ob der Bereich der Zahlen nur positiv oder auch negativ ist. Das Attribut *byteSize* gibt an, wieviele Byte zur Speicherung des Typs benötigt werden. Beispielsweise ist in Abbildung 3.3 ein Java-int deklariert.

Ein *FloatTypeDeclaration*-Element besitzt zusätzlich das Attribut *IEEEType*. Dieses Attribut repräsentiert durch einen der folgenden Werte die Genauigkeit der Fließkommazahl. Nach dem IEEE 754 Standard kommen folgende Werte in Frage: single-precision (32 bit) und double-precision (64 bit).

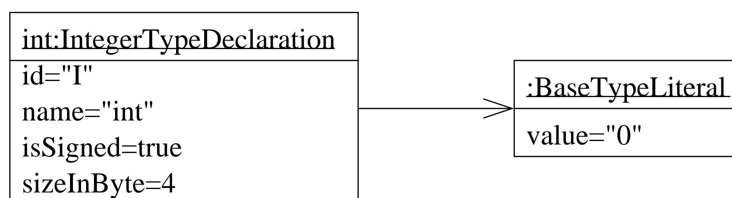


Abbildung 3.3: Externe Basistypen im GAST

Das Element *CharacterTypeDeclaration* repräsentiert Datentypen zur Darstellung einzelner Zeichen. Diese Datentypen variieren im benötigten Speicherbedarf (Attribut *sizeInByte*) und im Encoding (Attribut *encoding*), welches für die Darstellung verwendet wird.

Eine *FunctionTypeDeclaration* wird für Sprachen benötigt, die Funktionen als Werte erster Ordnung auffassen. Dies ist beispielsweise in der Sprache Lua der Fall (siehe [17] Kap. 2.2).

Die beiden verbleibenden Elemente, *BooleanTypeDeclaration* und *VoidTypeDeclaration*, repräsentieren boolesche Typen und den leeren Typ. Der boolesche Typ ist integraler Bestandteil des GASTs, so dass dieser immer deklariert werden muss. Benötigt wird er nicht nur als Typ zur Deklaration von Variablen und Funktionen, sondern auch als interner Typ für die Darstellung der Bedingungen in Kontrollflussanweisungen.

3.5.2 Deklaration von Funktionen und Konstanten

Funktionen und Konstanten werden über die GAST-Elemente *VariableDeclarationStatement* aus Abschnitt 3.9.2 und *FunctionDeclarationStatement* aus Abschnitt 3.9.3 deklariert. Ihre Bedeutung ist dabei völlig analog zu den entsprechenden Elementen im Programmbereich des GAST Modells.

3.5.3 Deklaration von externen, zusammengesetzten Datentypen

Externe, zusammengesetzte Datentypen werden, wie in Abschnitt 3.13 erläutert, in diesem Abschnitt durch die Elemente *PackageDeclaration* und *ClassDeclaration* deklariert.

3.5.4 Nachteile dieses Vorgehens

Das in diesem Abschnitt skizzierte Vorgehen hat neben dem Vorteil, die externen Deklarationen möglichst präzise zu charakterisieren, leider auch einen gravierenden Nachteil.

Dieser Nachteil des Verfahrens besteht in den fehlenden semantischen Informationen über die extern deklarierten Typen, Attribute und Methoden. Diese Informationen können nicht alleine aus den Informationen gewonnen werden, die in einem SAST vorhanden sind, da aus einem SAST nur syntaktische Informationen über die externen Elemente extrahiert werden können.

Problematisch sind beispielsweise Funktionen zum Extrahieren eines Substrings. Diese haben meistens - unter anderem - zwei ganzzahlige Parameter. Der erste dient als Angabe der Startposition, der zweite Parameter ist implementierungsabhängig und entweder eine absolute Angabe der Endposition des gewünschten Strings oder die Länge.

Ein weiteres Beispiel ist die Funktion *Node.replaceChild(Node p1, Node p2)*. Entgegen den Erwartungen, dass $p1$ durch $p2$ ersetzt wird, ist $p1$ der neue Knoten und $p2$ der ursprüngliche Knoten (entnommen der Java Online Dokumentation [3]). Dies wäre nur durch Analyse des Codes in Erfahrung zu bringen. Dieser steht aber nicht immer zur Verfügung.

Die genaue Semantik einer Funktion geht also aus ihrer Signatur nicht hervor. Ebenso kann der Wert von Konstanten nicht in der Liste der externen Deklarationen abgelegt werden.

3.6 Der Typ *Type* als Basistyp aller Typen

Der Typ *Type* wird als Ausgangsbasis für alle Typen im GAST verwendet. Dies geschieht analog zum Typ **Object** in Java oder vergleichbaren Konstruktionen in anderen Sprachen.

Dies dient unter anderem zur Umsetzung von heterogenen Datenstrukturen in ungetypten Sprachen, da diesen intern meist über gemeinsame Typen in der Vererbungshierarchie gelöst werden.

3.7 Referenzen auf Typen

Typen werden im GAST über die eindeutige ID des Typs referenziert. Als Element des GAST Modells wird dies wie in Abbildung 3.4 repräsentiert. Das Attribut *ref* bezieht sich hierbei immer auf die ID einer Typdeklaration.

Referenzen auf Typen aus dem Bereich für externe Deklarationen müssen nicht gesondert behandelt, da auch diese Typen über eine ID verfügen und GAST-weit zugreifbar sind.

Soll ein generischer Typ in einer Deklaration verwendet werden, so ist eine Ausprägung notwendig. Diese Ausprägungen der generischen Typen können auf drei sich ausschließende Arten durchgeführt werden:

1. Ausprägung mit Ausdrücken. Diese müssen bereits zum Zeitpunkt der Übersetzung ausgewertet werden können.

2. Ausprägungen mit Typen.

3. Ausprägung mit einem Wildcardtyp. Dieser kann mit weiteren Einschränkungen belegt werden. Diese Einschränkungen werden durch ein *WildcardBoundary* angeben. Das Attribut *kind* gibt dabei die Art der Beschränkung an. Die Konstante *none* bedeutet, dass keine Beschränkung existiert, *extends* beschränkt den unbekannten Typ nach oben durch den angegebenen Typ. Analog beschränkt *super* den unbekannten Typ von oben durch den angegebenen Typ.

Die genaue Semantik der Ausprägung mit Ausdrücken ist dem C++ Standard zu entnehmen, die genaue Semantik der Wildcardtypen ist [15] zu entnehmen.

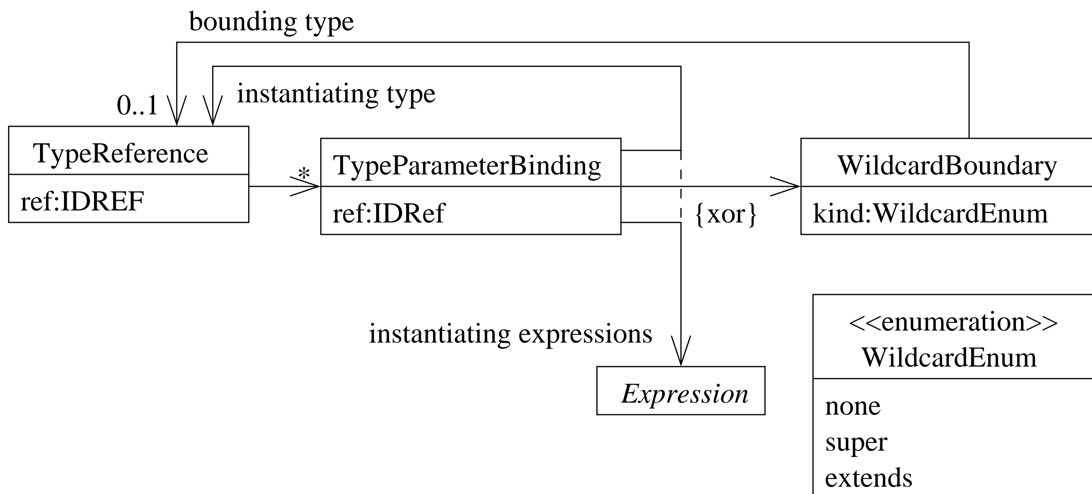


Abbildung 3.4: Details zur Typreferenz im GAST

3.8 Basisdatenstrukturen

Fast alle Programmiersprachen besitzen (mindestens) eine komplexere Datenstruktur, die im Sprachumfang verankert ist. Im GAST soll nun versucht werden, universelle Datenstrukturen zu definieren, die möglichst alle Anforderungen

erfüllen und dennoch vergleichsweise leicht in allen Sprachen zu implementieren sind bzw. für die möglichst in allen Sprachen Unterstützungen existieren.

Es bezeichne im Folgenden $\mathbb{T}$ die Menge aller Typen, d.h. alle Elemente, die direkt oder indirekt von der Klasse *Type* erben.

In den GAST werden folgende drei Datenstrukturen als Basisdatenstrukturen integriert und wie in Abbildung 3.5 in die Typhierarchie eingefügt.

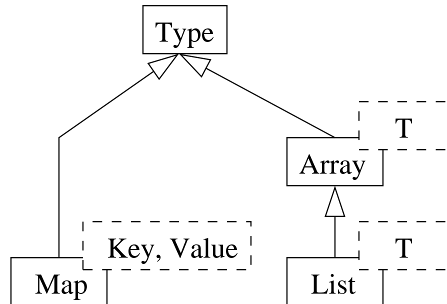


Abbildung 3.5: Einbettung der Basisdatenstrukturen in die Typhierarchie

Dieses Klassendiagramm ist aus Implementierungssicht zu lesen, d.h. eine GAST-List hat mehr Attribute und Funktionen als ein GAST-Array. Deshalb wird in Diagramm 3.5 die dargestellte Reihenfolge gewählt.

3.8.1 GAST-Array

Ein *GAST-Array* $\mathbb{A}_T$ ist eine Sequenz von Werten eines Typs fester Länge. Diese Sequenz besitzt einen Startindex, der standardmäßig den Wert “0” besitzt. In den Spezifikationen der Funktionen wird ein Array mit folgender abkürzender Schreibweise dargestellt: $(w_s, w_{s+1}, \ldots, w_{s+l-1})$.

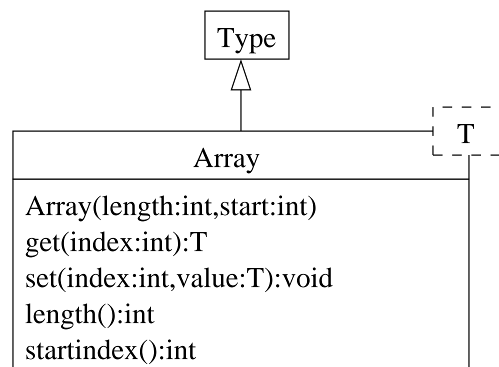


Abbildung 3.6: Funktionalität der Basisdatenstruktur Array

Als Operationen stellt der Typ *GAST-Array* Folgendes zur Verfügung:

- Konstruktor: $constructor : \mathbb{T} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{A}_T$ definiert durch: $constructor : \{T, l, s\} \mapsto a$ erzeugt ein neues Array a mit Einträgen vom Typ T der festen Länge l und dem Startindex s. Man erhält das Array $(t_s, t_{s+1}, \ldots, t_{s+l-1})$, wobei t_i vom Typ T ist und auf den dem Typ zugeordneten Initialisierungswert gesetzt wird.

- Lesen: $get : \mathbb{A}_T \times \mathbb{Z} \rightarrow T$, definiert durch: $get : \{a, i\} \mapsto a_i$ falls $s \leq i < s+l$ und für den Fall $i < s$ oder $s + l \leq i$ wird ein Fehler ausgelöst.

- Schreiben: *set:* $\mathbb{A}_T \times \mathbb{Z} \times T \rightarrow \mathbb{A}_T$, definiert durch: $set : \{a, i, t\} \mapsto a$ und setzt, im Fall $s \leq i < s + l$, $a_i = t$ und ist undefiniert sonst. Auch hier wird im für den Fall $i < s$ oder $s + l \leq i$ ein Fehler ausgelöst.

- Abfragen der Länge: *length:* $\mathbb{A}_T \rightarrow \mathbb{Z}$ liefert die feste Länge l des Arrays.

- Abfragen des Startindexes: *startindex:* $\mathbb{A}_T \rightarrow \mathbb{Z}$ liefert den Startindex s des Arrays.

Im GAST Modell werden mehrdimensionale Arrays über Arrays von Arrays dargestellt.

Die erzeugten Einträge werden vorinitialisiert. Dies geschieht bei Basistypen (siehe Abschnitt 3.5.1) mit dem dort als "initialization" assoziierten Element *Expression*. Bei komplexen Typen (Klassen, Schnittstellen und Aufzählungen) wird mit dem *NilLiteral* vorinitialisiert.

3.8.2 GAST-List

Eine *GAST-List* $\mathbb{L}_T$ ist eine Sequenz von Werten eines Typs mit variabler Länge. Wie das Array besitzt auch die Liste einen Startindex s. In den Spezifikationen der Funktionen wird eine Liste mit folgender abkürzender Schreibweise dargestellt: $[w_s, w_{s+1}, \ldots, w_{s+l-1}]$.

Folgende Operationen werden vom Typ *GAST-List* zur Verfügung gestellt:

- Konstruktor: $constructor : \mathbb{T} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{L}_T$ definiert durch: $constructor : \{T, c, s\} \mapsto a$ erzeugt eine neue Liste mit Einträgen vom Typ T der vorinitialisierten Länge c und dem Startindex s. Man erhält die Liste $[t_s, t_{s+1}, \ldots, t_{s+c-1}]$ wobei t_i mit $i \in \{s, \ldots s + c - 1\}$ vom Typ T ist.

- Lesen: Lesender Zugriff funktioniert analog zum lesenden Zugriff auf Arrays.

- Schreiben: Schreibender Zugriff funktioniert analog zum schreibenden Zugriff auf Arrays.

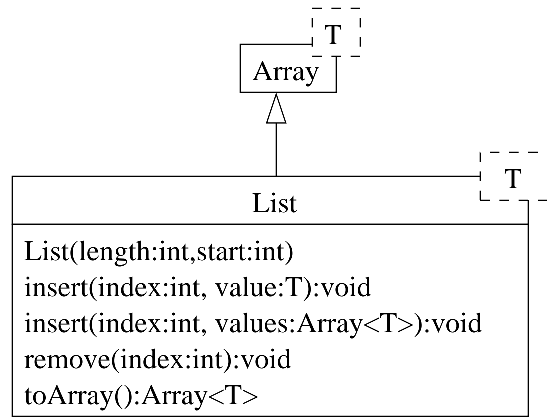


Abbildung 3.7: Funktionalität der Basisdatenstruktur List

- Einfügen von einzelnen Elementen: *insert:* $\mathbb{L}_T \times \mathbb{Z} \times T \rightarrow \mathbb{L}_T$ ist definiert durch: $insert : \{l, i, t\} \mapsto l$ und fügt den Wert t an der Stelle i ein. Alle weiteren Elemente werden um eine Position nach rechts verschoben. Dabei entsteht aus der Liste $[l_s, l_{s+1}, \ldots, l_{i-1}, l_i, l_{i+1}, \ldots, l_{s+c-1}]$ die modifizierte Liste $[l_s, l_{s+1}, \ldots, l_{i-1}, t, l_i, l_{i+1}, \ldots, l_{s+c-1}]$. Nach dem Einfügen des Elementes t an der Stelle i befinden sich alle Elemente mit Positionen $p > i$ an der Stelle $p + 1$. Die Länge der Liste wird um 1 erhöht. Wird an der Stelle $s - 1$, also am Anfang der Liste eingefügt, so wird der Startindex s um 1 verringert. Im Fall $i < s - 1$ oder $s + c < i$ ist der Zugriff ein Fehler.

- Einfügen von Elementen aus einem Array: *insert:* $\mathbb{L}_T \times \mathbb{Z} \times \mathbb{A}_T \rightarrow \mathbb{L}_T$ ist definiert durch: $insert : \{l, i, a\} \mapsto l$ und fügt die Werte aus dem **GAST-Array** a an der Stelle i ein und verschiebt alle weiteren Elemente um $length$ Positionen nach rechts. Dabei wird aus der Liste $[l_s, l_{s+1}, \ldots, l_{i-1}, l_i, l_{i+1}, \ldots, l_{s+c-1}]$ die Liste $[l_s, l_{s+1}, \ldots, l_{i-1}, a_1, \ldots, a_{a.length}, l_i, l_{i+1}, \ldots, l_{s+c-1}]$. Die Länge der Liste wird um $a.length()$ erhöht. Im Fall $i < s$ oder $s + c < i$ ist der Zugriff ein Fehler.

- Entfernen von einzelnen Elementen: *remove:* $\mathbb{L}_T \times \mathbb{Z} \rightarrow \mathbb{L}_T$ ist definiert durch: $remove : \{l, i\} \mapsto l$ und entfernt den Eintrag l_i falls $s \leq i < s + l$. Dabei entsteht aus der Liste $[l_s, l_{s+1}, \ldots, l_{i-1}, l_i, l_{i+1}, \ldots, l_{s+c-1}]$ die Liste $[l_s, l_{s+1}, \ldots, l_{i-1}, l_{i+1}, \ldots, l_{s+c-1}]$. Die Länge der Liste verringert sich um 1. Im Fall $i < s$ oder $s + c \leq i$ wird die Liste nicht verändert und es wird ein Fehler ausgelöst.

- Entfernen von Elementen: *remove:* $\mathbb{L}_T \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{L}_T$ ist definiert durch: $remove : \{l, i, j\} \mapsto l$ und entfernt die Einträge l_k mit $k \in \{i, \ldots, j\}$ falls $s \leq k < s{+}c$. Dabei entsteht aus der Liste $[l_s, l_{s+1}, \ldots, l_{i-1}, l_i, l_{i+1}, \ldots, l_{s+c-1}]$ die Liste $[l_s, l_{s+1}, \ldots, l_{i-1}, l_{j+1}, \ldots, l_{s+c-1}]$. Die Länge der Liste wird dabei

um $j - i$ verringert. Für den Fall, dass eine der beiden Grenzen außerhalb des Listenbereichs liegt, wird die Liste nicht verändert und es wird ein Fehler ausgelöst.

- Extrahieren einer Teilliste: *sublist:* $\mathbb{L}_T \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{L}_T$ ist definiert durch: $\mathit{sublist} : \{l, i, j\} \mapsto l$ und speichert die Einträge l_k mit $k \in \{i, \ldots, j\}$, falls $s \leq k < s + c$, in einer neuen Liste $[l_i, l_{i+1}, \ldots, l_{j-1}, l_j]$. Für den Fall, dass eine der beiden Grenzen außerhalb des Listenbereichs liegt, wird die Liste nicht verändert und es wird ein Fehler ausgelöst.

- Aneinanderhängen zweier Listen gleichen Typs: *concat:* $\mathbb{L}_T \times \mathbb{L}_T \rightarrow \mathbb{L}_T$ ist definiert durch $\mathit{concat} : \{a, b\} \mapsto [a_{s_a}, \ldots a_{s_a+c_a-1}, b_{s_b}, \ldots, b_{s_b+c_b-1}]$. Man erhält also eine Liste, die die Elemente der Listen a und b enthält sowie den Startindex s_a und die Länge $c_a + c_b$ besitzt.

- Abfragen der Länge: *length:* $\mathbb{L}_T \rightarrow \mathbb{Z}$ liefert die momentane Länge c der Liste.

- Abfragen des Startindexes: *startindex:* $\mathbb{L}_T \rightarrow \mathbb{Z}$ liefert den Startindex s der Liste.

- Transformation in ein **GAST-Array**: *toArray:* $\mathbb{L}_T \rightarrow \mathbb{A}_T$ definiert durch $\mathit{toArray} : l \mapsto (l_s, l_{s+1}, \ldots, l_{s+c-1})$. Dies ermöglicht die Iteration über die Einträge der Liste mit Hilfe der *foreach*-Schleife.

Eine Vorinititialisierung wird analog zur Vorinititialisierung der der Basisdatenstruktur *Array* vorgenommen.

3.8.3 GAST-Map

Eine *GAST-Map* $\mathbb{M}_{S,W}$ ist eine Zuordnung von Schlüsseln des Typs S auf Werte des Typs W. Mathematisch gesehen ist eine Abbildung A eine Teilmenge des kartesischen Produkts $S \times W$ ($A \subset S \times W$). Dabei wird aus einer Menge von Schlüsseln S_D in eine Menge von Werten W_D abgebildet. Jeweils einem Schlüssel $s \in S_D$ wird genau ein Wert $w \in W_D$ zugeordnet.

Als Operationen stehen dabei folgende zur Verfügung:

- Konstruktor: *constructor* : $S \times W \rightarrow \mathbb{M}_{S,W}$ erzeugt eine neue Abbildung $\mathit{Abb} : S \rightarrow W$. Diese besitzt eine leere Schlüsselmenge S_D und folglich eine leere Wertemenge W_D.

- Abfragen von Zuordnungen: *get:* $\mathbb{M}_{S,W} \times S \rightarrow W$ ist definiert durch $\mathit{get} : \{m, s\} \mapsto w$ für $w = \mathit{Abb}(s)$, falls $s \in S_D$. Sonst löst diese Operation einen Fehler aus.

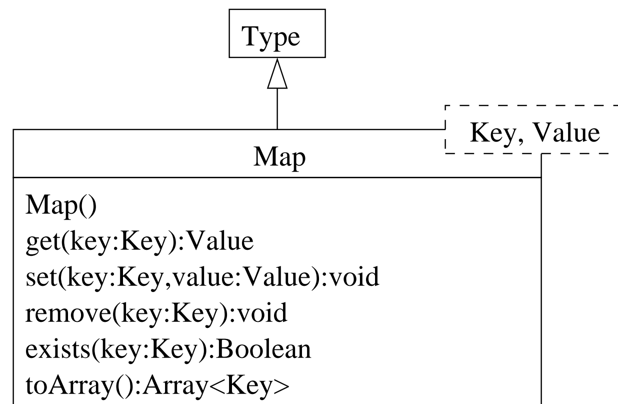


Abbildung 3.8: Funktionalität der Basisdatenstruktur Map

- Verändern von Zuordnungen: *set:* $\mathbb{M}_{S,W} \times S \times W \rightarrow \mathbb{M}_{S,W}$, definiert durch: $set : \{m, s, w\} \mapsto m$.

  Hierbei werden zwei Fälle unterschieden:

  1. $s \in S_D$: In diesem Fall wird der Wert der Abbildung $Abb(s) := w$ festgelegt.
  2. $s \notin S_D$: In diesem Fall wird die Schlüsselmenge erweitert $S_D := S_D \cup \{s\}$ und der Wert der Abbildung $Abb(s) := w$ festgelegt.

- Entfernen von Zuordnungen: *remove:* $\mathbb{M}_{S,W} \times S \rightarrow \mathbb{M}_{S,W}$, definiert durch: $remove : \{m, s\} \mapsto m$. Hierbei werden zwei Fälle unterschieden:

  1. $s \in S_D$: In diesem Fall wird s aus der Schlüsselmenge entfernt $S_D := S_D \setminus \{s\}$.
  2. $s \notin S_D$: In diesem Fall wird nichts verändert.

- Abfragen, ob Schlüssel bereits enthalten ist: *exists:* $\mathbb{M}_{S,W} \times S \rightarrow \mathbb{B}$ definiert durch $exists : \{m, s\} \mapsto \{true, false\}$ liefert *true*, falls $s \in S_D$ und sonst *false*.

- Auslesen der Schlüsselmenge: *toArray:* $\mathbb{M}_{S,W} \rightarrow Array(S)$ definiert durch $toArray : \mathbb{M}_{S,W} \mapsto (s_1, s_2, \ldots, s_n)$. Dies ermöglicht die Iteration über die Einträge der Abbildung. Eine spezifische Reihenfolge der Elemente wird nicht garantiert.

Die IDs aus Tabelle 3.2 werden bei Referenzierungen der Basisdatenstrukturen verwendet.

Typ	id
Array	A
List	L
Map	M

Tabelle 3.2: IDs der Basisdatenstrukturen

3.8.4 Zusammenfassung der Basisdatenstrukturen

Abschließend zu den Definitionen der Datenstrukturen soll nun kurz untersucht werden, welche Programmiersprachen welche Datenstrukturen bereits im Sprachstandard beinhalten (und nicht über eine Standardbibliothek bereit stellen) und in wie weit diese Strukturen dann über die drei GAST Strukturen abgedeckt sind. Die Auswahl der Sprachen ist ein Querschnitt durch [16].

- **C** (siehe [16] Kapitel 2.3) bietet neben Arrays mit fester Länge und festem Startindex 0 so genannte *struct*s zu Generierung zusammengesetzter Datentypen an. Die *struct*s können im GAST durch Klassen umgesetzt werden, benötigen also keine zusätzlichen Elemente im GAST. Die Arrays können im GAST durch GAST-Arrays dargestellt werden, der Startindex ist hierbei auf "0" zu setzen.

- **C++** (siehe [16] Kapitel 3.3 und 3.6) erweitert die Möglichkeiten von C um die Bildung von Klassen, also Einheiten von Daten und Programmanweisungen. Diese werden im GAST ebenfalls durch Klassen dargestellt.

- **Pascal** (siehe [16] Kapitel 6.3) bietet Arrays, die nicht zwingend 0-indiziert sein müssen, sondern beliebige ganzzahlige Indexbereiche haben dürfen. Dies unterstützt der GAST durch die Möglichkeit GAST-Arrays mit Startindex zu erzeugen.

  Weiter existieren Mengen (sets), diese können durch eine GAST-Map simuliert werden. Dabei werden die Einträge der Pascal-Menge als Schlüssel einer GAST-Map vom Typ der Pascal-Menge auf $\mathbb{B}$ (Boolescher Typ) aufgefasst.

  Records aus Pascal können im GAST durch Klassen repräsentiert werden, die keine Methoden besitzen.

- **Perl** (siehe [16] Kapitel 15.2 und 15.5) bietet dynamische Listen und Hashes. Perl-Listen können durch GAST-Listen und Perl-Hashes durch GAST-Maps repräsentiert werden.

- **Lua** (siehe [17] Kapitel 2.2) bietet nur eine Datenstruktur, genannt *Table*. Diese ist ein assoziatives Array und kann je nach Verwendung im Programm

durch ein GAST-Array, eine GAST-Liste oder eine GAST-Map repräsentiert werden.

- **Python** (siehe [16] Kapitel 18.2) bietet ebenfalls geordnete Mengen von Werten (*Tupel*) und assoziative Felder (*Dictionaries*) an. Diese können durch GAST-Arrays und GAST-Maps dargestellt werden.

Heterogene Datenstrukturen existieren nur in ungetypten Sprachen. Deshalb erfolgt eine Abbildung auf eine homogene Struktur von Objekten des Typs "Type".

Sollen in getypten Sprachen heterogene Datenstrukturen dargestellt werden, so werden diese durch Klassen modelliert. Die Einträge der Tupel werden zu Attributen der Klasse mit entsprechenden Typen und Namen.

3.9 Anweisungen

Anweisungen besitzen im GAST Sprungmarken, die für die *Break*- (siehe Abschnitt 3.9.13) und *Continue*-Anweisung (siehe Abschnitt 3.9.14) notwendig sind. Die Sprungmarken werden im Attribut *label* im Element *Statement* abgelegt.

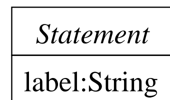


Abbildung 3.9: Modell für ein Label im GAST

Die Nassi-Shneiderman-Diagramme (siehe [31]) wurden zur sprachunabhängigen Beschreibung von Algorithmen entwickelt und haben sich für diesen Zweck in der Praxis bewährt. Analog zu den Nassi-Shneiderman-Diagrammen beinhaltet die Definition des GASTs die folgenden Anweisungen.

3.9.1 Block

Ein Block ist eine Folge von Befehlen, die linear abgearbeitet werden (siehe Abbildung 3.10). Dies entspricht den Blöcken aus bekannten Sprachen und bildet eine Einheit für die Sichtbarkeit von Variablen.

Ein Block kann in Codeform wie folgt dargestellt werden:

```
1 begin
2   statement_1;
3   //weitere Statements
4   statement_n;
5 end
```

Listing 3.3: Beispiel für einen Block

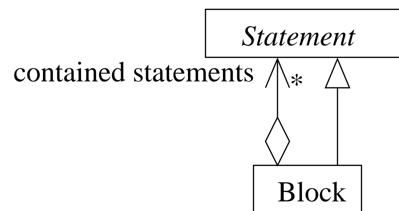


Abbildung 3.10: Modell für den Block im GAST

3.9.2 Deklaration von Variablen

Die Deklaration einer Variablen wird im GAST wie in Abbildung 3.11 dargestellt.

Variablen sind im GAST typisiert, besitzen also eine Referenz auf einen Typ. Um ungetypte Sprachen umsetzen zu können, wird in solchen Sprachen auf den Typ *Type* zugegriffen, der in der Hierarchie der Typen als Basisklasse für alle nativ in der Sprache angelegten Typen dient.

Die Sichtbarkeit der definierten Variablen ist auf den lokalen Bereich beschränkt, d.h. die lokale Sequenz von Befehlen, in der die Variable definiert wurde. Sichtbarkeitsbereiche sind verschachtelt, d.h. auch in inneren Anweisungen ist die Variable sichtbar, solange sie nicht durch eine weitere Deklaration gleichen Namens verdeckt wird.

Um Anfragen im GAST nach Variablendeklarationen zu erleichtern, wird eine projektweit eindeutige *id* vergeben. Diese kann verwendet werden, um bei Zugriffen auf Variablen sofort die entsprechende Deklaration zu bekommen.

Modifizierende Angaben wie 'final' aus Java oder 'const' aus C werden im GAST über so genannte *Modifier* modelliert. Diese werden in Abschnitt 3.13.6 näher erläutert.

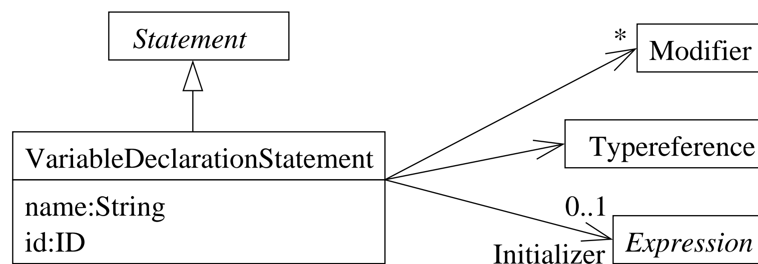


Abbildung 3.11: Modell für eine Variablendeklaration im GAST

Drei Variablendeklarationen sieht man im nächsten Codefragment.

```
Float diameter;
Float area;
Float pi = 3.14;
```

Listing 3.4: Beispiel für Variabledeklarationen

3.9.3 Deklaration von Funktionen

Im GAST werden Funktionendeklarationen wie in Abbildung 3.12 dargestellt.

Es wird im GAST nicht zwischen Funktionen und Prozeduren unterschieden. Dies ist z.B. in Pascal und verwandten Sprachen der Fall. Stattdessen wird der Rückgabetyp auf einen Typ ohne Wertebereich (meist void) gesetzt (Siehe zu voidartig die Deklaration der externen Basistypen in Abschnitt 3.5.1).

Ebenfalls werden im GAST keine Standardwerte für Parameter zugelassen. Diese müssen durch Angabe entsprechender Literale bei allen Aufrufen der Funktion ersetzt werden.

Der GAST unterstützt bei Funktionsaufrufen Polymorphie, d.h. Funktionen mit verschiedenen Parameterlisten werden wie verschiedene Funktionen gehandhabt. Dies entspricht dem Verhalten von Java.

Ebenso werden generische Funktionen vom GAST unterstützt. Dies geschieht über die optional vorhandenen *TypeParameterDeclaration*-Elemente, welche im Abschnitt 3.13.4 genau eingeführt werden.

Eine Funktion wird im GAST eindeutig über ihre *id* referenziert, der Name wird für besser lesbaren Code gespeichert. Die optionalen Modifier-Elemente werden in Abschnitt 3.13.6 beschrieben. Der Rumpf ist in Verbindung mit dem Modifier *abstract* nicht anzugeben. Durch eine *TypeReference* wird der Rückgabewert der Funktion deklariert. Der Typ muss hierfür im Instanzdokument referenzierbar sein.

Parameter werden durch das Element *ParameterDeclaration* eingeführt und über ihre IDs im GAST referenziert. Diese Elemente besitzen neben dem Attribut *name*, der eindeutigen ID zwei weitere Attribute. Das boolesche Attribut *isVarArg* repräsentiert hierbei einen Parameter mit variabler Vielfachheit. Dieses Attribut darf nur beim letzten Parameter der Funktion auf den Wert <u>*true*</u> gesetzt sein. Das boolesche Attribut *isFrozen* repräsentiert hierbei einen konstanten Parameter.

Im Beispiel wird eine generische Funktion 'average' deklariert, die den Mittelwert beliebig vieler Parameter berechnet und einen Wert vom generischen Typ *T* zurückliefert. Die Parameter sind vom Typ T.

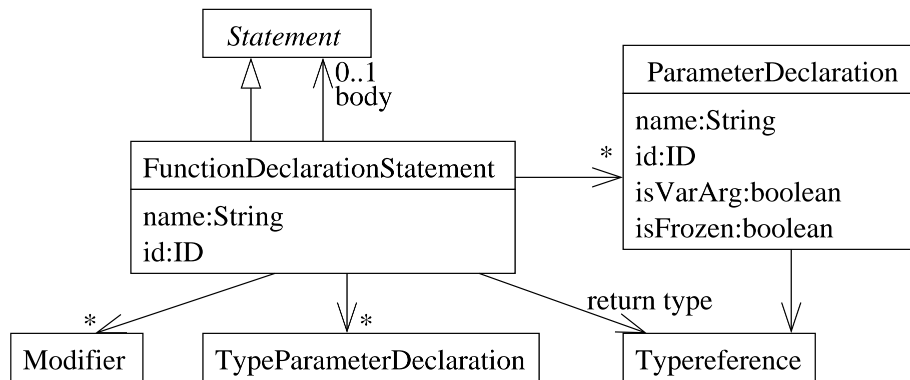


Abbildung 3.12: Modell für eine Anweisung zur Deklarieren von Methoden

```
1 function<T> average(T... args):T
2 begin
3   T result;
4   Integer length = args.length();
5   for(Integer i = 0;i<length;i = i+1)
6     result = result + args.get(i);
7   return result / length;
8 end
```

Listing 3.5: Beispiel einer Funktionsdeklarationanweisung

3.9.4 Ausdrucksanweisung

Ausdrucksanweisungen kapseln Ausdrücke und verwerfen den Wert des Ausdrucks. Die Umsetzung im GAST sieht man in Abbildung 3.13.

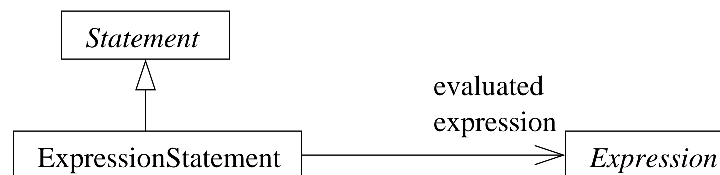


Abbildung 3.13: Modell für die Ausdrucksanweisung des GASTs

Beispiele für Ausdrucksanweisungen sieht man im nächsten Codebeispiel.

```
1 diameter = 10;
2 area = pi * (diameter / 2)^2 ;
```

Listing 3.6: Beispiel für Ausdrucksanweisungen

3.9.5 Zweifachauswahl

Eine Zweifachauswahl ist eine Alternative im Code und entspricht weitgehend einer *If*-Anweisung mit optionalem *Else*-Teil. Ihre Umsetzung im GAST sieht man in Abbildung 3.14.

Die Bedingung muss einen booleschen Wert besitzen. Dieser Ausdruck wird hierbei genau einmal evaluiert. Das Ergebnis entscheidet dann, welche Anweisung ausgeführt wird. Im Falle *true* ist dies der *Then*-Teil, ansonsten der optionale *Else*-Teil.

Eine einfache Auswahl kann durch Weglassen des optionalen *Else*-Teils erreicht werden, eine mehrfache Auswahl wird über geschachtelte Zweifachauswahlen erreicht. Aus diesem Grund fehlt die Möglichkeit, die Zweifachauswahl durch *ElseIf*-Teile zu erweitern.

Das Problem der Darstellung des 'dangling else' tritt im GAST aufgrund der Schachtelung der Elemente nicht auf. Wird ein SAST in den GAST transformiert, so muss diese Transformation das Problem der 'dangling else' lösen, z.B. durch Einfassen in Blöcke.

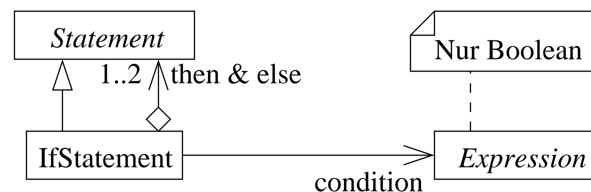


Abbildung 3.14: Modell für die Zweifachauswahl des GASTs

Als Erweiterung von Listing 3.6 und als Beispiel für eine Zweifachauswahl dient folgendes Codefragment:

```
1 if(diameter > 0)
2   print(area);
3 else
4   print("error");
```

Listing 3.7: Beispiel für eine Zweifachauswahl

Ein Beispiel für eine Einfachauswahl:

```
1 if(diameter > 0)
2   print(area);
```

Listing 3.8: Beispiel für eine Einfachauswahl

3.9.6 Fallauswahl

Eine Fallauswahl entspricht einer *Switch-Case*-Anweisung in anderen Sprachen.

Sie bietet sich an, wenn mehrere Bedingungen überprüft werden sollen, die jeweils einen gemeinsamen Ausdruck auf der linken Seite eines binären Operators besitzen. Dieser Ausdruck wird hierbei nur einmal ausgewertet. Danach werden die Fallbedingungen überprüft. Die Zusammenhänge sieht man in Abbildung 3.15.

Möglichkeiten für Fallbedingungen sind (betrachtet wird jeweils der Wert des Ausdrucks, der im *SwitchStatement* angegeben wurde):

- *Value*
  Hier wird der betrachtete Wert und der Wert des Ausdruck des *CaseClause* auf Wertgleichheit geprüft.

- *Range*
  Hier wird geprüft, ob der betrachtete Wert im Bereich des Ausdruck des *CaseClause* enthalten ist.

- *Default*
  Dieser Teil wird nur dann ausgeführt, wenn kein anderer *CaseClause* ausgeführt wurde.

Im GAST kann über das boolesche Attribut *isFallThrough* das Verhalten der Fallauswahl beeinflusst werden. Ist das Attribut auf *true* gesetzt, so wird nach durchlaufen einer Anweisung die Fallauswahl nicht beendet, sondern es wird mit der nächsten Answeisung fortgefahren, solange bis eine Unterbrechung durch *break* oder *continue* eintritt. Ist das Attribut auf *false* gesetzt, so wird nach der Abarbeitung einer Anweisung der Fallauswahl diese sofort verlassen.

In Abschnitt 4.4 wird ein Algorithmus angegeben, der eine Transformation auf dem GAST beschreibt, die **fall-through switch-statements** in **non-fall-through switch-statements** umsetzt.

Jede der Fallbedingungen hat eine optionale Anweisung, die abgearbeitet wird, wenn der entsprechende Fall eingetreten ist. Wird keine Fallbedingung erfüllt, so wird der optionale *default*-Teil abgearbeitet. Ist dieser nicht vorhanden, so werden keine Anweisungen ausgeführt.

Eine primitive Art zu zählen zeigt folgendes Beispiel für die Fallauswahl:

```
1 Integer input = 3;
2 switch(input, isFallThrough=false)
3   case 1:
4     print("eins");
5   case 2..5:
6     print("eine Hand voll");
7   case 6..10:
8     print("zwei Haende voll");
```

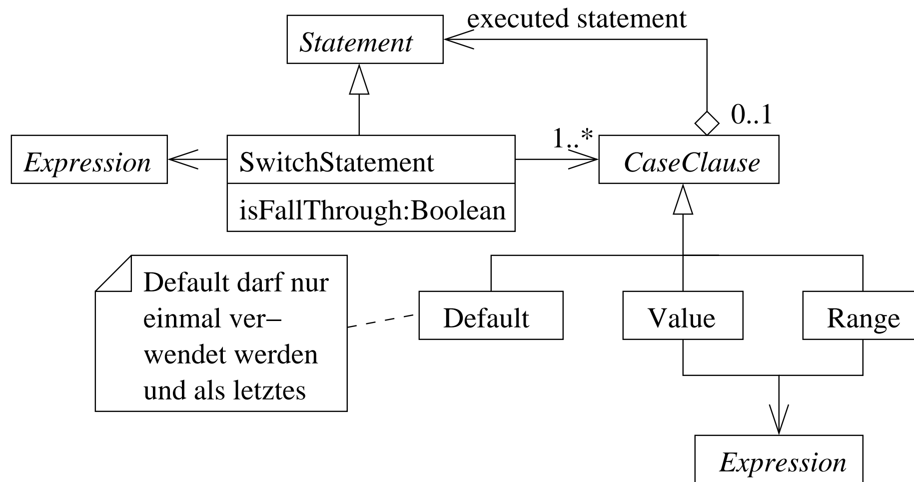


Abbildung 3.15: Modell für die Fallauswahl des GASTs

```
9    default:
10       print("viele");
11 end
```

Listing 3.9: Beispiel für eine Fallauswahl

Es wird also “eine Hand voll” ausgegeben.

3.9.7 For-Schleife

Diese Form der Schleife besitzt einen Initialisierungsteil, einen Bedingungsteil und einen Berechnungsteil. Jeder dieser Teile ist optional, wie man Abbildung 3.16 entnehmen kann.

Der Initialisierungsteil (*Initalisation*) wird vor dem ersten Durchlaufen der Schleife einmalig ausgeführt. Darin dürfen entweder eine Reihe von Variablen deklariert oder eine Folge von Ausdrücken ausgewertet werden.

Die Bedingung wird jeweils vor dem Durchlaufen der Schleife geprüft und muss ein Ergebnis von booleschem Typ besitzen. Die Anweisung im Rumpf der Schleife wird ausgeführt, solange die Bedingung auf *true* evaluiert wird.

Nach dem Abarbeiten des Rumpfs wird der Berechnungsteil der Schleife ausgeführt. Dieser besteht aus einer Reihe von Ausdrücken. Diese werden in ihrer Reihenfolge ausgewertet.

Das *VariableDeclaration*-Element besitzt die Attribute *id* und *name*, deren Bedeutung analog zu denen beim *VariableDeclarationStatement* ist. Gleiches gilt für die *Expression* und die *TypeReference*.

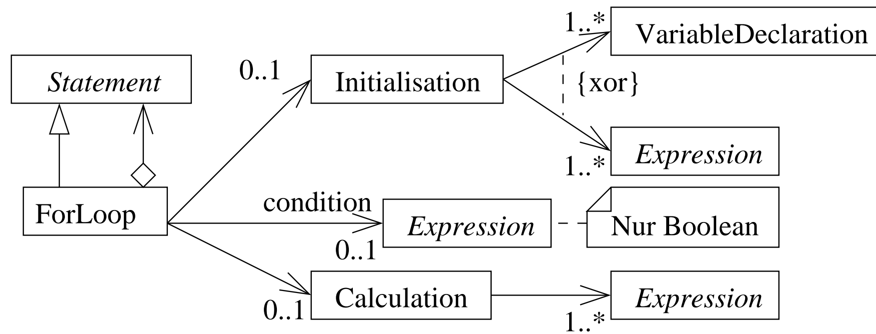


Abbildung 3.16: Modell für die zählende Schleife des GASTs

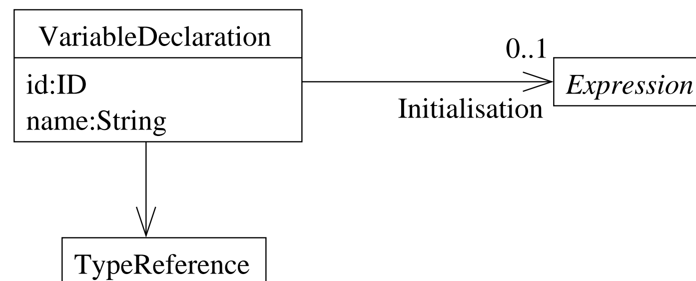


Abbildung 3.17: Modell für die interne Deklaration einer Variablen

Folgendes Codefragment berechnet die Fakultät von n:

```
1 //Eingabe ist die Zahl n
2 Integer fak = 1;
3 for (Integer i = 1; i < n; i = i+1)
4   fak = fak * i;
```

Listing 3.10: Beispiel für eine zählende Schleife

3.9.8 Foreach-Schleife

Die iterierende Schleife wird verwendet, um Operationen elegant auf alle Elemente einer Datenstruktur anwenden zu können oder um über Bereiche zu iterieren. Siehe dazu die Definition des Bereichsoperators in Abschnitt 3.11.4.

Die Voraussetzung, um über Datenstrukturen iterieren zu können, ist eine Methode *toArray()*, die ein GAST-Array mit allen enthaltenen Element der Datenstruktur zurückliefert. Alle im GAST definierten Datenstrukturen erfüllen diese Voraussetzung bereits.

Erster Schritt bei der Ausführung dieses Schleifentyps ist die Deklaration einer Iterationsvariablen. Werte des zu iterierenden Arrays müssen an diese Iterations-

variable zuweisbar sein. Zusätzlich darf diese Variable keine Initialisierung besitzen. Der iterierte Ausdruck muss, wie oben beschrieben, eine Funktion *toArray()* besitzen. Diese wird einmalig aufgerufen und das Array zwischengespeichert. Anschließend durchläuft eine interne Indexvariable dieses Array, speichert jeweils ein Element in der Iterationsvariablen und führt dann die enthaltene Anweisung aus.

Die genaue Darstellung der Foreach-Schleife im GAST sieht man in Abbildung 3.18. Die Erklärungen zum *VariableDeclaration*-Element finden sich in Abschnitt 3.9.7.

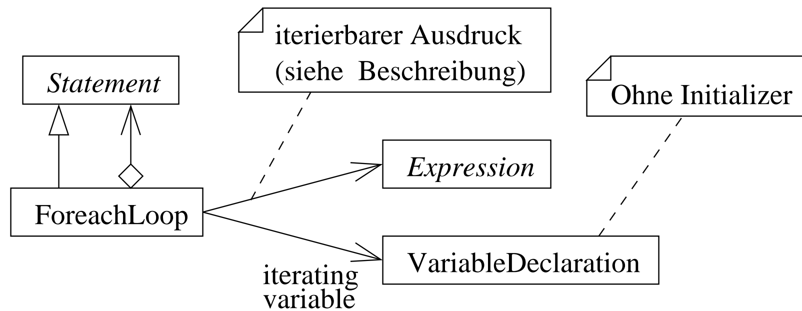


Abbildung 3.18: Modell für die iterierende Schleife des GASTs

Folgendes Codefragment iteriert über eine GAST-Map und gibt die Schlüssel-Wert-Paare aus:

```
1 //Eingabe ist die Map m einer ungetypten Sprache
2 foreach(Type key in m)
3 begin
4   Type value = m.get(key);
5   // # ist der Stringkonkatenationsoperator
6   print(key # "=>" # value);
7 end
```

Listing 3.11: Beispiel für eine iterierende Schleife

3.9.9 Bedingungsgesteuerte Schleife

Die bedingungsgesteuerte Schleife besitzt eine Bedingung, die zu einem booleschen Wert auswertbar sein muss, und eine Anweisung als Rumpf (siehe Abbildung 3.19).

Durch ein boolesches Attribut (*headControlled*) wird festgelegt, ob die Schleife kopfgesteuerter oder fußgesteuerter Art ist.

Im Fall der kopfgesteuerten Schleife wird bereits vor dem ersten Durchlauf des Rumpfs die Bedingung geprüft. Der Rumpf wird also eventuell nicht durchlaufen.

Im Fall der fußgesteuerten Schleife wird hingegen der Rumpf mindestens einmal durchlaufen und erst anschließend die Bedingung geprüft. In beiden Fällen wird, wenn die Schleife auf den Wert <u>*false*</u> evaluiert, abgebrochen.

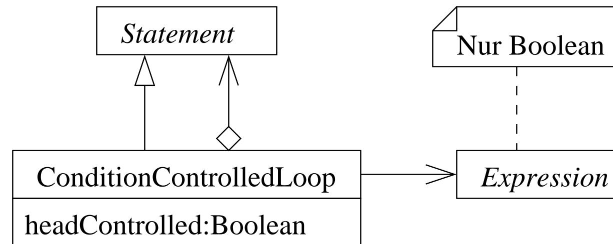


Abbildung 3.19: Modell für die bedingungsgesteuerte Schleife des GASTs

Eine fußgesteuerte Schleife wird wie im nächsten Codefragment dargestellt:

```
1 do
2   statement;
3 while(condition);
```

Listing 3.12: Beispiel für eine fußgesteuerte Schleife

Folgendes Codefragment berechnet den größten gemeinsamen Teiler von a und b in einer kopfgesteuerten Schleife und gibt diesen aus:

```
1 //Eingabe: 2 ganze Zahlen a,b
2 while(b > 0) do
3 begin
4   Integer r = a % b; //lokale Variable
5   a = b;
6   b = r;
7 end
8 print(a);
```

Listing 3.13: Beispiel für eine kopfgesteuerte Schleife

3.9.10 Throw-Statement

Diese Anweisung drückt aus, dass ein Fehler aufgetreten ist. Der Ausdruck aus dem GAST Modell ist hierbei eine Repräsentation des aufgetretenen Fehlers (siehe Abbildung 3.20).

Wird mit dieser Anweisung ein Fehler ausgelöst, so wird die normale Abarbeitung der Anweisungen unterbrochen und in der Hierarchie der Aufrufe nach oben gegangen, bis eine *Monitor*-Anweisung (siehe Abschnitt 3.9.11) den Fehler abfängt. Existiert keine solche Anweisung, so endet das Programm sofort.

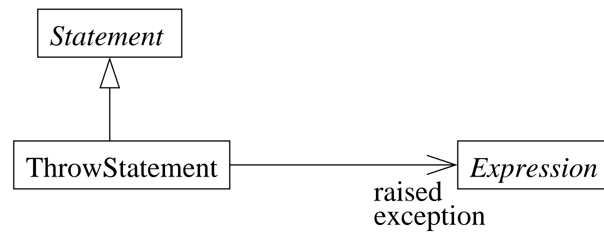


Abbildung 3.20: Modell für eine Anweisung zum Auslösen von Fehlern im GAST

Folgendes Codefragment zeigt eine Anwendung des Throw-Statements:

```
1 function division(Float a, Float b):Float
2   if(b == 0)
3     throw "Division by zero";
4   else
5     return a/b;
6 end
```

Listing 3.14: Beispiel für ein Throw-Statement

3.9.11 Monitored-Statement

Überwachte Anweisungen werden benutzt, um Fehlersituationen in Anweisungen zu behandeln. Die genaue Struktur wird in Abbildung 3.21 dargestellt.

Die *CatchBlock*-Elemente in der Anweisung behandeln spezifische aufgetretene Fehler, der optionale *FinallyBlock* wird immer ausgeführt. Einzige mögliche Ausnahme ist ein unkontrolliertes Programmende.

Tritt in der überwachten Anweisung kein Fehler auf, so wird direkt, falls vorhanden, die Anweisung aus dem *FinallyBlock* ausgeführt. Ist diese nicht vorhanden, wird direkt die nachfolgende Anweisung ausgeführt.

Tritt in der überwachten Anweisung hingegen ein Fehler auf, so wird die Liste der Bewacher der Reihenfolge nach abgearbeitet und anhand des Typs des Fehlers geprüft, ob der Bewacher der Zuständige ist. Diese Prüfung wird durch einen Test der Typgleichheit mit Hilfe *instanceof* Operators durchgeführt.

Entspricht kein Typ dem entsprechenden Bewacher, so wird der Fehler in der Aufrufhierarchie nach oben durchgereicht.

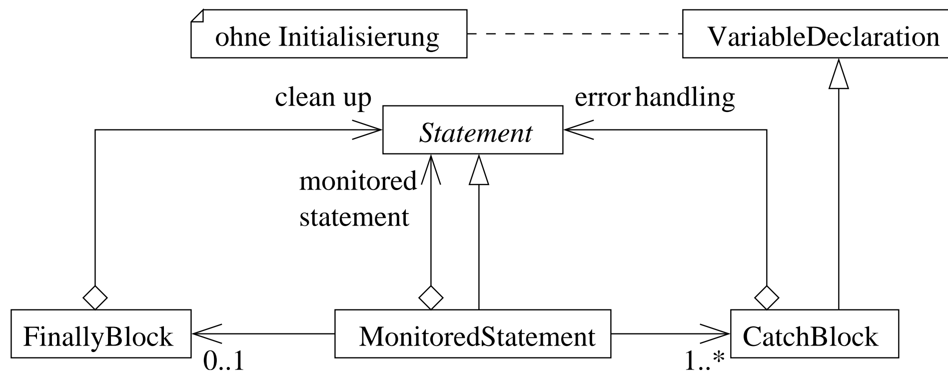


Abbildung 3.21: Modell für überwachte Anweisungen im GAST

Folgendes Codefragment erweitert Listing 3.14 und zeigt eine Anwendung des Monitored-Statement:

```
1 monitor
2   division(4,0);
3 catch(String exception)
4   print(exception);
```

Listing 3.15: Beispiel für ein Throw-Statement

3.9.12 Return-Statement

Eine Return-Anweisung beendet die aktuelle Funktion und liefert als Wert des Funktionsaufrufs den Wert des angegebenen Ausdrucks zurück (siehe dazu Abbildung 3.22).

Zugelassen ist eine Return-Anweisung nur als einzelne Anweisung oder als letzte Anweisung in einem Block, da Anweisungen nach dem Return nicht ausgeführt würden. Dadurch wird der aus Java bekannte "unreachable code" verhindert (siehe [5]).

Der Typ des Ausdrucks der Return-Anweisung muss dem Rückgabetyp zuweisbar sein.

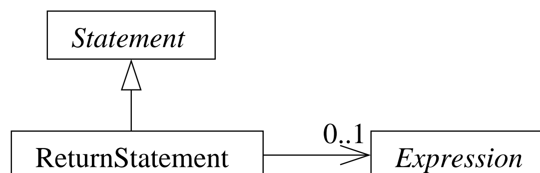


Abbildung 3.22: Modell für die Return Anweisung im GAST

3.9.13 Break-Statement

Break-Anweisungen besitzen im GAST immer eine Marke als Ziel (siehe Abbildung 3.23). Diese gibt an, welches Statement vorzeitig verlassen, bzw. nach welcher Anweisung fortgesetzt werden soll.

Die Positionen der Break-Anweisung im Code ist auf einzelne Anweisungen direkt ohne Block oder als letzte Position im Block beschränkt, da der Code im Block nach dem Break nicht erreichbar wäre. Durch diese Definition soll verhindert werden, dass der aus Java bekannte "unreachable code" entsteht (siehe [5]).

Auch die Position der Zielmarke ist bei einer Break-Anweisung beschränkt. Sie darf nur innerhalb einer Funktion oder Methode verwendet werden und muss sich im GAST auf einer direkten Verbindung zwischen dem *BreakStatement* und dem Anfang der Deklaration der ersten umfassenden Methode bzw. Funktion befinden. Das bedeutet, dass eine Prüfung dieser Bedingung einem Abschreiten der Achse der Elternelemente im Baum entspricht. Weiterhin können, syntaktisch gesehen, nur solche Anweisungen unterbrochen werden, die weitere Anweisungen in Form einer Komposition enthalten können.

Tritt ein Break auf, so wird über die Marke das zu verlassende Statement bestimmt und dessen Nachfolgeanweisung ausgeführt.

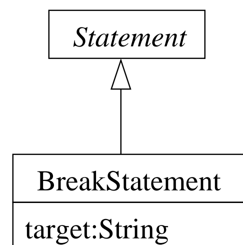


Abbildung 3.23: Modell für die Break Anweisung im GAST

Folgendes Codefragment demonstriert die Verwendung der Break-Anweisung in Kombination mit einer If-Anweisung:

```
1  Integer a = 5;
2  Integer b = 6;
3  l1: if(a == 5)
4  begin
5    print("a");
6    if(b == 6)
7      break l1;
8    print("b");
9  end
10 print("finish");
```

Listing 3.16: Beispiel für ein Break-Statement

Dieses Fragment gibt zunächst “a” und dann “finish” aus, überspringt also Zeile 7.

3.9.14 Continue-Statement

Continue-Anweisungen besitzen, wie Break-Anweisungen auch, im GAST immer eine Zielmarke (siehe Abbildung 3.24). Diese gibt nun an, bei welchem Schleifenkonstrukt vorzeitig die nächste Iteration beginnen soll.

Die Position der Continue-Anweisungen im Code ist wie bei den Break-Anweisungen beschränkt, siehe Abschnitt 3.9.13.

Die gleichen Einschränkungen wie für die Marken der Break-Anweisungen existieren auch bei den Continue-Anweisungen. Zusätzlich sind diese jedoch nur an For-Schleifen, Foreach-Schleifen und bedingungsgesteuerten Schleifen zulässig.

Tritt in einer solchen Schleife ein *ContinueStatement* auf, so wird die Anweisung der entsprechenden Schleife abgebrochen und die Kontrolle an die Schleifenanweisung übergeben. Das bedeutet, dass bei For- und bei Foreach-Schleifen jeweils der nächste Iterationsschritt gestartet wird.

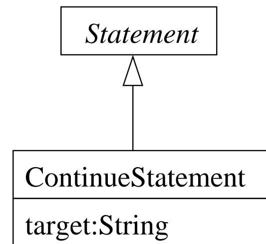


Abbildung 3.24: Modell für die Break Anweisung im GAST

Folgendes Codefragment demonstriert die Verwendung der Continue-Anweisung in Kombination mit einer For-Schleife:

```
1 Integer prod = 1;
2 l1: for(Integer i = 0; i < 10; i = i + 1)
3 begin
4   if(i % 2 == 0)  //nur ungerade Zahlen aufmultiplizieren
5     continue l1;
6   prod = prod * i;
7 end
```

Listing 3.17: Beispiel für ein Continue-Statement

3.10 Ausdrücke

Ausdrücke besitzen, anders als Anweisungen, einen Wert und somit auch einen Typ. Deshalb werden Ausdrücke, wie in Abbildung 3.25 dargestellt, mit einer TypeReference gekoppelt.

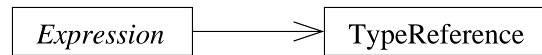


Abbildung 3.25: Modell für Ausdrücke im GAST

3.10.1 Literale

Literale sind im Code stehende Konstanten der Basistypen und der Basisdatentypen.

Literale der Basistypen werden im GAST durch ein *BasetypeLiteral*-Element repräsentiert (siehe Abb. 3.26). Diese haben als Attribute den Wert des Literals als String abgelegt. Eine Ausnahme dabei sind die Literale für Zeichenketten (Strings) oder für einzelne Zeichen. Diese werden jeweils ohne die begrenzenden Zeichen abgelegt, da diese sprachspezifisch sind. Dazu werden beispielsweise die " um die String und die ' um die einzelnen Zeichen entfernt.

Literale für Funktionstypen im herkömmlichen Sinne existieren im GAST nicht, da Funktionen entweder über das Element *FunctionDeclarationStatement* (siehe Abbschnitt: 3.9.3) oder über das Element *FunctionDeclaration* (siehe Abbschnitt: 3.10.6) deklariert werden.

Eine Ausnahme ist das *BooleanLiteral*; Dieses repräsentiert einen booleschen Wert in einer normalisierten Darstellung. Das bedeutet, die beiden einzigen Literale sind *true* und *false*. Die individuellen Literale der einzelnen SAST Modelle müssen in diese beiden GAST-Literale konvertiert werden.

Das *NilLiteral* ist ein Literal, welches zu allen Typen zuweisungskompatibel ist. Da ein Wert hierfür ausreichend ist, wird dieser nicht angegeben. Das Literal bedeutet hierbei, dass kein gültiger Wert in der Variablen etc. gespeichert ist.

Das Element *TypeLiteral* beschreibt ein Literal, welches einen Typ direkt in den enthaltenen Ausdruck einbringt. Dies wird im GAST verwendet, um den binären Operator **instanceof** zu definieren. Die Referenz in diesem Element verweist dabei auf eine Typdeklaration innerhalb des GAST Dokuments.

```
1 1510                    //Integer Literal
2 5.7                      //Float Literal
3 true                     //Boolean Literal
4 "Hallo Welt!"           //String Literal
5 nil                      //Nil Literal
```

Listing 3.18: Beispiel für Literale im GAST (I)

Literale für die Basisdatentypen *Array* und *Map* werden durch die Elemente *ArrayLiteral* und *MapLiteral* dargestellt (siehe Abb. 3.26). Arrayliterale beginnen immer ab Startwert "0". Diese Literale verweisen auf weitere *Expression*-Elemente, um den Inhalt der Datenstruktur zu bestimmen. Die Einträge eines Map-Literals treten immer paarweise als Schlüssel und Wert auf.

```
1 {1,2,3,4,5}                //Array Literal
2 { "world" -> "welt" }      //Map Literal
```

Listing 3.19: Beispiel für Literale im GAST (II)

Listen besitzen keine eigenen Literale. Stattdessen werden ihre Literale durch jeweils ein Array-Literal und einen Funktionsaufruf "insert" verwendet.

```
1 (new List<int>()).insert({1,2,3,4,5});
```

Listing 3.20: Initialisierung einer Liste

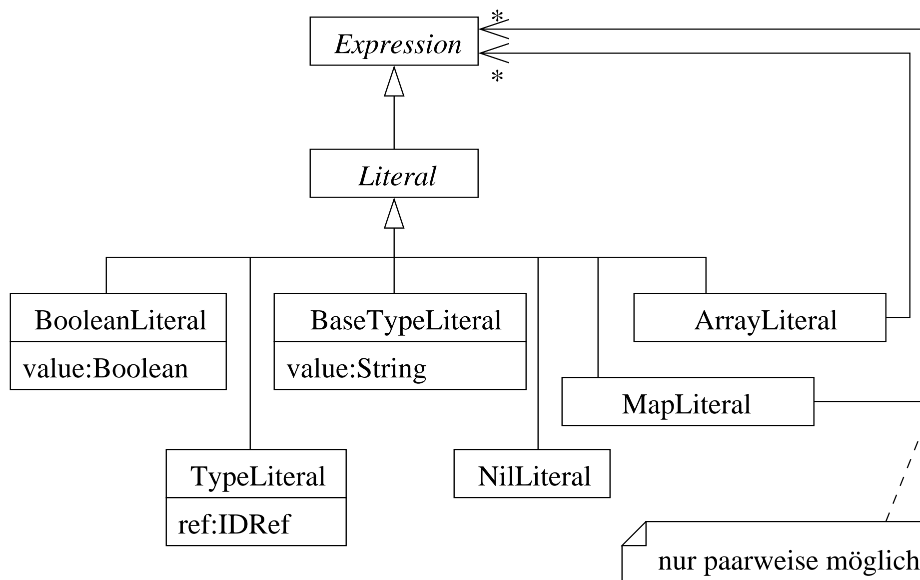


Abbildung 3.26: Modell für ein Literal im GAST

3.10.2 Zugriffe auf Variablen

Zugriffe auf Variablen werden gemäß Abbildung 3.28 durchgeführt.

Variablen werden im GAST nicht über ihren Namen, sondern über eine projektweit eindeutige *id* angesprochen. So werden keine qualifizierenden Elemente benötigt. Diese werden durch die Transformationen beim Schreiben in den Quellcode erzeugt.

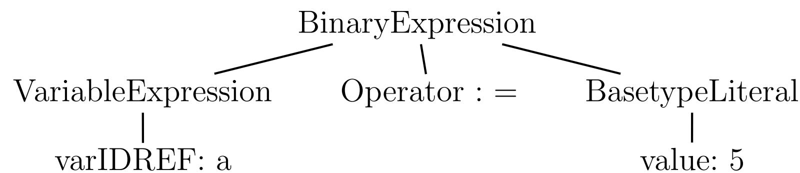


Abbildung 3.27: Der GAST für eine Zuweisung an eine Variable

Zuweisungen an Variablen werden durch die Kombination einer *BinaryExpression* (Operator "=") und einem *VariableAccess*-Element durchgeführt. Beispielsweise wird das Codefragment `a = 5` zu dem Baum aus Abbildung 3.27.

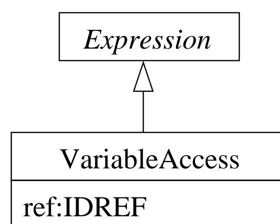


Abbildung 3.28: Modell für einen Zugriff auf eine Variable im GAST

3.10.3 Unäre Ausdrücke

Unäre Ausdrücke kombinieren einen Operator mit einem Ausdruck. Die Semantik des Ausdrucks ergibt sich hierbei aus der Kombination des Ausdrucks und des Operators. Der Operator wird der Liste in Kapitel 3.11 entnommen. Die unären Ausdrücke werden im GAST wie in Abbildung 3.29 dargestellt.

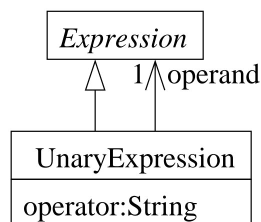


Abbildung 3.29: Modell für einen unären Ausdruck im GAST

3.10.4 Binäre Ausdrücke

Die Operatoren für binäre Ausdrücke finden sich in Abschnitt 3.11. Der Wert des Ausdrucks wird durch Anwendung des Operators auf die beiden Operanden berechnet. Die binären Ausdrücke werden im GAST wie in Abbildung 3.30 dargestellt.

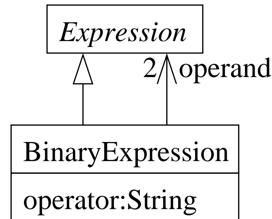


Abbildung 3.30: Modell für einen binären Ausdruck im GAST

3.10.5 Der Ternäre Ausdruck

Der ternäre Ausdruck besitzt als drei *Expression*s als Kindelemente. Der erste Ausdruck muss auf einen booleschen Wert evaluierbar sein, die beiden anderen auf beliebige Typen.

Die Funktionsweise des ternären Ausdrucks ist analog zu bekannten Programmiersprachen. Der erste Ausdruck, die Bedingung, gibt an welcher der beiden weiteren Ausdrücke ausgewertet wird. Wird der erste Ausdruck zu *true* ausgewertet, so wird der erste der beiden weiteren Ausdrücke ausgewertet, ansonsten der zweite. Der jeweilige Wert wird dann auch als Wert des ternären Ausdrucks zurück geliefert.

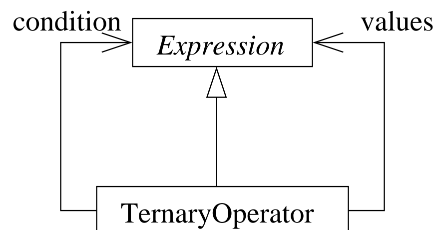


Abbildung 3.31: Modell für den ternären Ausdruck im GAST

3.10.6 Funktionsdeklaration

Funktionsdeklarationen sind Ausdrücke von einem Funktionstyp. Dieser existiert nicht in allen, bevorzugt aber in funktionalen Sprachen, die Funktionen als Werte

erster Ordnung begreifen.

Funktionen werden also bei Variablendeklarationen als Initialisierer oder als rechte Seite bei Zuweisungen verwendet. Anonyme Funktionen oder Funktionsparameter sind ebenso möglich. Eine ID für die Funktion ist in einem solchen Fall nicht vorgesehen. Dafür wird bei Funktionsaufrufen die ID der entsprechenden Variablen verwendet, welche über einen qualifizierenden Ausdruck angegeben wird. Die weiteren Details zu Funktionsdeklarationen sind in Abschnitt 3.9.3 beschrieben. Das Element *TypeParameterDeclaration* wird in Abschnitt 3.13.4 genau eingeführt.

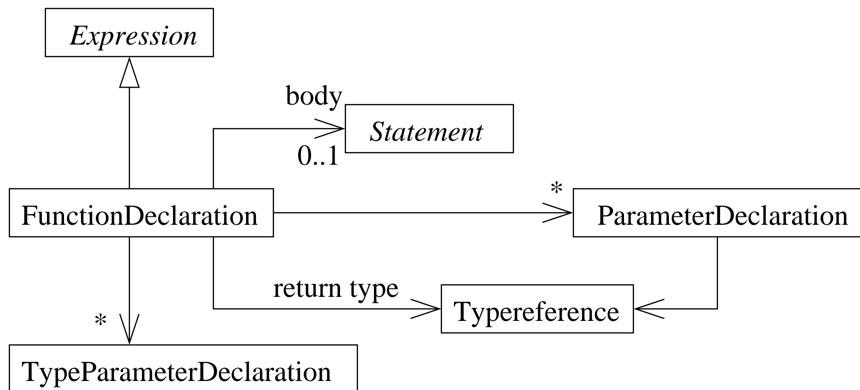


Abbildung 3.32: Modell für eine Funktionsdefinition im GAST

Im Beispiel wird, um die Anwendung des Funktionstyps zu demonstrieren, eine generische Funktion deklariert, die den Mittelwert beliebig vieler Parameter berechnet und einen Wert vom generischen Typ *T* zurückliefert. Die Parameter sind vom Typ T. Diese Funktion wird dann der Variablen 'average' zugewiesen (um sie im nächsten Beispiel aufrufen zu können).

```
1 average = function<T>(T... args):T
2 begin
3   T result;
4   for(Integer arg : args)
5     result = result + arg;
6   return result / length;
7 end
```

Listing 3.21: Beispiel einer Funktionsdeklaration

3.10.7 Funktionsaufruf

Funktionsaufrufe im GAST erfolgen über eine Referenz auf die ID der Funktion. Diese wird bei Funktionsdeklarationen als Anweisung (siehe Abschnitt. 3.9.3)

direkt im Element mitangegeben, bei Funktionsdeklarationen als Ausdruck (siehe Abschnitt. 3.10.6) wird die ID des Zuweisungsziels genommen.

Der optionale qualifizierende Ausdruck wird benötigt, um Methoden auf Objekten aufzurufen.

Bei Funktionsaufrufen werden zuerst die Parameter ausgewertet und dann ein Aufruf mit Call-By-Reference vorgenommen. Soll ein Aufruf mit Call-By-Value simuliert werden, so müssen alle die entsprechenden Ausdrücke in der Parameterliste kopiert werden. Der Rückgabewert der Funktion wird als Wert des Ausdrucks genommen.

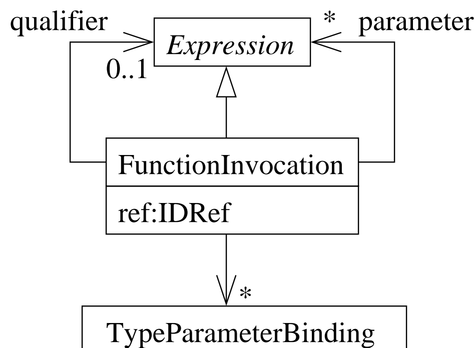


Abbildung 3.33: Modell für einen Funktionsaufruf im GAST

Um generische Funktionen aufrufen zu können, müssen alle Typparameter durch *TypeParameterBinding*-Elemente gebunden sein. Dies ist entweder durch Angabe von Typen oder Angabe von Ausdrücken, deren Wert zum Zeitpunkt der Übersetzung feststehen, möglich.

Im Beispiel wird die Funktion 'average' aus Listing 3.21 mit den Parametern 1 und 2 aufgerufen und das Ergebnis der Variablen 'avrg' zugewiesen.

```
1 Integer avrg = average<Integer >(1,2);
```

Listing 3.22: Beispiel eines Funktionsaufrufs

3.11 Operatoren

In der Definition des GASTs findet sich eine feste Menge an Operatoren. Dies ist eine Auswahl an Operatoren aus gängigen Programmiersprachen. Auf die Angabe einer Operatorhierarchie im GAST kann verzichtet werden, da die Hierarchie der Operatoren bereits durch die Verschachtelung der Elemente gegeben ist. Diese Verschachtelung wird einfach aus dem SAST übernommen. Ebenso verhält es sich mit der Assoziativität der Operatoren.

In der Notation in Codeform wird in komplexeren Fällen deshalb eine ausführliche Klammerung vorgenommen.

3.11.1 Logische Operatoren

Logische Operatoren erwarten in Abhängigkeit ihrer Stelligkeit (unär oder binär) entweder ein oder zwei Ausdrücke von booleschem Typ. Ihr Ergebnis ist ebenfalls immer ein Ausdruck vom booleschen Typ.

- logisches Nicht(**!**)
  Stelligkeit: unär
  Wirkung: Invertiert den angegebenen booleschen Wert von *true* nach *false* und umgekehrt.

- logisches Und(**&&**)
  Stelligkeit: binär
  Wirkung: Verknüpft die beiden booleschen Werte durch eine Und-Funktion, dabei wird der zweite Operand nicht ausgewertet, wenn bereits der erste den Wert *false* hatte.

- logisches Oder(||)
  Stelligkeit: binär
  Wirkung: Verknüpft die beiden booleschen Werte durch eine Oder-Funktion. Dabei wird der zweite Operand nicht ausgewertet, wenn bereits der erste den Wert *true* hatte.

- logisches Xor(˜)
  Stelligkeit: binär
  Wirkung: Verknüpft die beiden booleschen Werte durch eine Exklusiv-Oder-Funktion. Dabei werden beide Operatoren ausgewertet.

Werden neben den short-circuit Operatoren || und **&&** ihre elementweisen Pendants benötigt, so müssen diese durch Funktionen ersetzt werden (Siehe dazu auch Abs. 4). In Java existieren sowohl short-circuit (z.B. **&&**) als auch die elementweisen Operatoren (z.B. **&**). Die Bezeichnungen “short-circuit” und “elementwise” stammen von [26].

3.11.2 Relationale Operatoren

Relationale Operatoren sind immer binäre Operatoren und liefern immer einen Ausdruck vom booleschen Typ zurück.

Folgende Operatoren stellen dabei keine Ansprüche an die Typen der Parameter, sie können deshalb Ausdrücke beliebigen Typs verarbeiten.

- Wertgleichheit(==)
  Wirkung: Liefert den booleschen Wert *true*, falls beide Operanden vom gleichen Typ sind und den gleichen Wert repräsentieren, bzw. wenn alle Elemente einer komplexeren Datenstruktur jeweils den gleichen Wert repräsentieren. Ansonsten wird *false* geliefert.

- Wertungleichheit(**!=**)
  Wirkung: Liefert den booleschen Wert *true*, falls beide Operanden unterschiedliche Werte repräsentieren. Dies ist der Fall, wenn sie unterschiedlichen Typen angehören, oder gesetzt den Fall, dass sie gleichen Typs sind, wenn die Werte verschieden sind. Ansonsten wird *false* geliefert.

- Identität(===)
  Wirkung: Liefert den booleschen Wert *false*, falls die Operanden unterschiedlichen Typs sind. Ansonsten wird bei Basistypen ein Test auf Wertgleichheit durchgeführt und bei Objekten wird geprüft, ob es sich um das gleiche Objekt handelt.

Folgende Operatoren setzen eine Ordnung auf den Typen der Operatoren voraus. Deshalb können nur Ausdrücke beliebiger geordneter Typen (ganze und reelle Zahlen sowie Zeichenketten, nicht jedoch boolesche Werte) verarbeitet werden.

- Kleiner(<)
  Wirkung: Liefert den booleschen Wert *true*, falls beide Operanden vom gleichen Typ sind und der erste Operand einen kleineren Wert repräsentiert. Ansonsten wird *false* geliefert.

- Größer(>)
  Wirkung: Liefert den booleschen Wert *true*, falls beide Operanden vom gleichen Typ sind und der erste Operand einen größeren Wert repräsentiert. Ansonsten wird *false* geliefert.

- Kleiner gleich(<=)
  Wirkung: Liefert den booleschen Wert *true*, falls beide Operanden vom gleichen Typ sind und der erste Operand einen kleineren Wert repräsentiert oder beide Operanden den gleichen Wert repräsentieren. Ansonsten wird *false* geliefert.

- Größer gleich(>=)
  Wirkung: Liefert den booleschen Wert *true*, falls beide Operanden vom gleichen Typ sind und der erste Operand einen größeren Wert repräsentiert oder beide Operanden den gleichen Wert repräsentieren. Ansonsten wird *false* geliefert.

3.11.3 Arithmetische Operatoren

Folgende Regel wird bei den binären arithmetischen Operatoren bezüglich der Unterscheidung des Ergebnistyps angewendet: Sind beide Operanden ganze Zahlen, so ist auch das Ergebnis der Operation ein ganzzahliger Typ, andernfalls ist das Ergebnis ein Fließkommatyp. Ausnahme hierbei ist die Operation Modulo, die nur für ganze Zahlen definiert ist.

- unäres Minus(-)
  Stelligkeit: unär
  Operanden: Ausdrücke mit Werten beliebiger numerischer Typen
  Ergebnis: ganze und reelle Zahlen (siehe obige Regel)
  Wirkung: führt eine Invertierung des Vorzeichens des Operanden durch und liefert das Ergebnis als Resultat zurück.

- Addition(+)
  Stelligkeit: binär
  Operanden: Ausdrücke mit Werten beliebiger numerischer Typen
  Ergebnis: ganze und reelle Zahlen (siehe obige Regel)
  Wirkung: führt eine Addition der beiden Operanden durch und liefert das Ergebnis als Resultat zurück.

- Subtraktion(-)
  Stelligkeit: binär
  Operanden: Ausdrücke mit Werten beliebiger numerischer Typen
  Ergebnis: Ausdruck numerischer Typen (siehe obige Regel)
  Wirkung: führt eine Subtraktion der beiden Operanden durch und liefert das Ergebnis als Resultat zurück.

- Multiplikation(*)
  Stelligkeit: binär
  Operanden: Ausdrücke mit Werten beliebiger numerischer Typen
  Ergebnis: Ausdruck numerischer Typen (siehe obige Regel)
  Wirkung: führt eine Multiplikation der beiden Operanden durch und liefert das Ergebnis als Resultat zurück.

- Division(/)
  Stelligkeit: binär
  Operanden: Ausdrücke mit Werten beliebiger numerischer Typen
  Ergebnis: Ausdruck numerischer Typen (siehe obige Regel)
  Wirkung: führt eine Division der beiden Operanden durch und liefert das Ergebnis als Resultat zurück.

- Modulo(%)
  Stelligkeit: binär
  Operanden: Ausdrücke mit Werten beliebiger ganzzahliger Typen
  Ergebnis: Ausdruck ganzzahliger Typen
  Wirkung: führt eine ganzzahlige Division der beiden Operanden durch und liefert als Resultat den Rest der Division zurück.

- Exponentiation(∧)
  Stelligkeit: binär
  Operanden: Ausdrücke mit Werten beliebiger numerischer Typen, mit der Einschränkung, dass der erste Operator (die Basis) nicht negativ sein darf.
  Ergebnis: Ausdruck numerischer Typen (siehe obige Regel)
  Wirkung: berechnet als Ergebnis $op_1^{op_2}$.

Auf die Verwendung des unären Plus, der Identität, wurde hier verzichtet, da dieser keine wertverändernde Semantik besitzt.

Ebenso wurde auf die in vielen Sprachen vorkommenden Inkrement- und Dekrementoperatoren verzichtet. Sie können, wie in Abschnitt 4.1 beschrieben, ersetzt werden.

3.11.4 Sonstige Operatoren

Die folgenden Operatoren fallen in keinen bisher aufgezählten Bereich, sind jedoch essentiell (cast und Zuweisung) oder in vielen Sprachen relevant und deshalb unverzichtbar.

- Zuweisungsoperator(=)
  Stelligkeit: binär
  Operanden: der erste Operand muss dabei Ziel einer Zuweisung sein können (*VariableAccess* oder *FieldAccess*), der zweite Ausdruck darf von beliebigem Typ sein.
  Ergebnis: beliebige Typen
  Wirkung: Der erste Operand muss zu einer Variablen evaluierbar sein, der zweite darf ein beliebiger Ausdruck sein. Der Wert der Zuweisung ist der Wert der Variablen nach der Zuweisung.

- Cast-Operator(**cast**)
  Stelligkeit: binär
  Operand: der erste Operand ist ein *TypeLiteral*, der zweite Operand ein Ausdruck beliebigen Typs
  Ergebnis: der Wert des Ausdrucks, der in den angegeben Zieltyp umgewandelt wurde
  Wirkung: Der Wert des Ausdrucks wird in den angegebenen Zieltyp gewandelt. Dabei sind sowohl up- als auch downcasts möglich.

- Typvergleich(**instanceof**)
  Stelligkeit: binär
  Operand: der erste Operand ein Ausdruck beliebigen Typs, der zweite Operand ein *TypeLiteral*
  Ergebnis: boolescher Wert
  Wirkung: prüft, ob der Ausdruck (erster Operand) eine Instanz des Typs, der durch den zwei Operanden gegeben wird, ist.

- Bereichsoperator(**...**)
  Stelligkeit: binär
  Operanden: Ausdrücke ganzzahligen Typs
  Ergebnis: Basisdatenstruktur
  Wirkung: liefert als Ergebnis eine Auflistung aller Werte zwischen op_1 und op_2 als Einträge eines Arrays $(op_1, \ldots, op_2)$ zurück.

- Stringkonkatenation(**@**)
  Stelligkeit: binär
  Operanden: Ausdrücke eines String Typs
  Ergebnis: String
  Wirkung: liefert als Ergebnis die Konkatenation der beiden String-Operanden zurück.

3.11.5 Weitere Operatoren

Operatoren, die nicht auf dieser Liste stehen, müssen im GAST simuliert werden. Möglich ist dies zum Beispiel durch die Verwendung von Funktionsaufrufen. Dies betrifft explizit den ternären Operator, der jedoch durch eine Umstrukturierung des Quellcodes simuliert werden muss.

3.12 Strukturierung des Quellcodes

Zur Strukturierung des Quellcodes werden so genannte Pakete verwendet. Auf eine Festlegung der Einteilung in Dateien und Verzeichnisse wird verzichtet, da eine solche Einteilung zu sprachspezifisch wäre.

Ein Programm im GAST besteht aus mindestens einer PackageDeclaration (siehe Abb. 3.34). Globale Fragmente des Ursprungs-SASTs außerhalb eines Pakets oder vergleichbarer Elemente werden in ein **Default-Package** abgebildet. Dieses Default-Package hat im GAST ein leeres Attribut für den Namen.

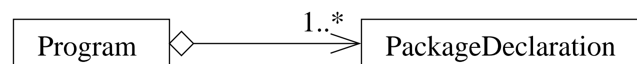


Abbildung 3.34: Modell für Programme im GAST

Pakete sind hierarchisch angeordnete Strukturen, die folgende Elemente des GASTs aufnehmen können:

- Paketdeklarationen
  Durch geschachtelte Pakete wird eine hierarchische Struktur geschaffen, die komplexere Systeme erlaubt.

- Anweisungen
  Dies ermöglicht die Umsetzung von Sprachen, die globale Anweisungen ohne besondere Startmethode zulassen (Lua, Python, etc.).

- Typdeklarationen
  Zu Typdeklarationen siehe Abschnitt 3.13.

Als UML-Klassendiagramm kann man Pakete wie in Abbildung 3.35 beschreiben.

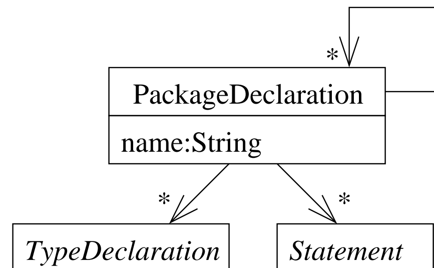


Abbildung 3.35: Modell für Pakete im GAST

Im GAST-Code werden Paketdeklarationen wie folgt notiert:

```
1 package GUI
2   //hier steht der Inhalt des Pakets
3 end
```

Listing 3.23: Beispiel einer Paketdeklaration

3.13 Erweiterung der Typauswahl

Die Object Management Group (OMG) hat mit der Unified Modeling Language (UML) (siehe [23]) eine Beschreibungssprache für objektorientierte Software geschaffen. In der UML sind unter anderem die Einführung von Klassen, Schnittstellen und Aufzählungen definiert. Im GAST werden diese sprachunabhängigen Möglichkeiten nun übernommen und so können neue Typen im GAST durch Typdeklarationen definiert werden.

3.13.1 Typdeklaration

Ausgangspunkt der Hierarchie (siehe Abbildung 3.36) für die Definition von Typen ist das abstrakte Element der *TypeDeclaration*. Hier werden die wichtigsten Eigenschaften von Typdeklarationen zentral definiert.

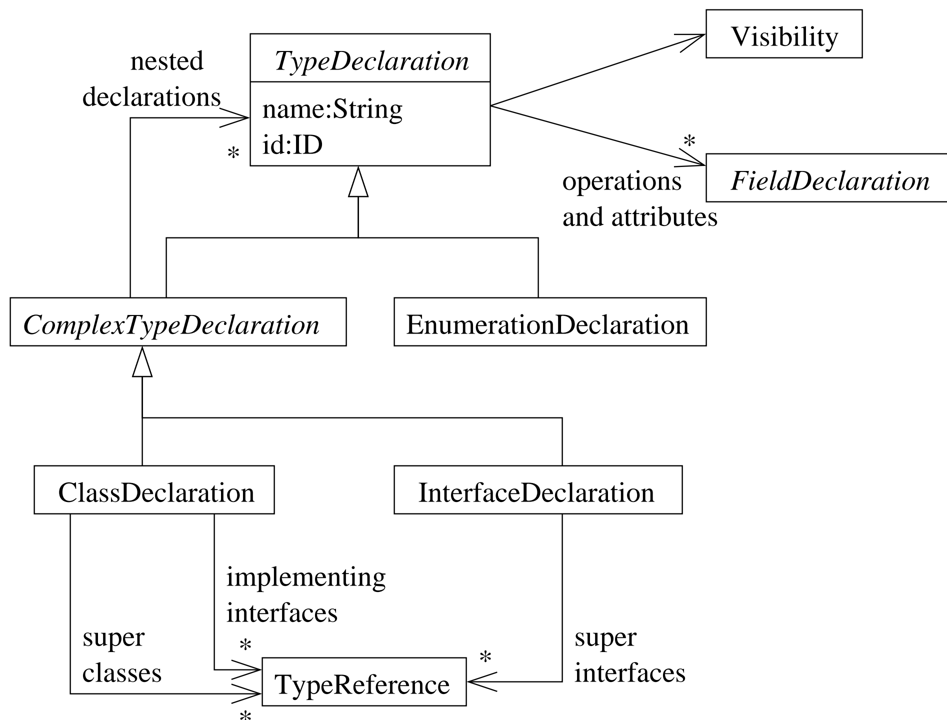


Abbildung 3.36: Die verschiedenen Typdeklarationen des GASTs

Typdeklarationen besitzen neben dem Typnamen eine projektweit eindeutige ID, die bei Typereferenzen verwendet wird.

Eine *TypeDeclaration* hat eine Sichtbarkeit (siehe Abbildung 3.49) bezüglich des umgebenden Pakets oder der umgebenden Typdeklaration. Ebenso besitzen die Operationen und Attribute eine Sichtbarkeit. Die Sichtbarkeiten werden im Abschnitt über die Deklaration von Klassen genauer erläutert. Die Felddeklarationen gleich im Anschluss.

3.13.2 Felddeklarationen und Feldzugriffe

Im GAST wird analog zur UML zwischen Attributen und Methoden unterschieden. Beide besitzen im GAST-Modell eine projektweit eindeutige *id* und einen Namen (gespeichert in *name*). Diese ID wird bei Zugriffen auf die entsprechenden Felder verwendet. Zusätzlich besitzen alle Felder eine Sichtbarkeit, ausgedrückt durch das Element *Visibility*. Optional hingegen ist die Angabe von *Modifier*-Elementen. Eine Übersicht sieht man in Abbildung 3.37.

Die *Attribute*-Elemente besitzen zusätzlich Assoziationen zu einer *TypeReference* und optional zu einer *Expression*. Die Typreferenz repräsentiert den Typ des Attributs und der Ausdruck einen möglichen Initialisierer.

```
1 attribute Float ratio = 0.25;
```

Listing 3.24: Beispiel einer Attributdeklaration

Elemente im GAST, die eine *Operation* darstellen, besitzen ein boolesches Attribut *isConstructor*. Zusätzlich existieren Verweise auf *ParameterDeclaration*-Elemente zur Spezifikation der Parameter. Der Rückgabetyp wird über ein *Type-Reference*-Element dargestellt, der Rumpf über ein optionales *Statement*. Generische Operationen werden über *TypeParameterDeclaration*-Elemente ermöglicht. Diese werden in Abschnitt 3.13.4 erklärt.

```
1 operation square(Float a):Float
2   return a^2;
3 end
```

Listing 3.25: Beispiel für eine Deklaration einer Operation

Konstruktoren sind spezielle Operationen, die durch ein auf *true* gesetztes Attribut *isConstructor* markiert sind. *Modifier*-Elemente sind an Konstruktoren nicht erlaubt, sie sind aber implizit statische Methoden, die als Rückgabetyp den deklarierenden Typ zurückliefern.

```
1 operation constructor(Float a):Klasse
2   // Initialisierung der Klasse
3 end
```

Listing 3.26: Beispiel für eine Konstruktordeklaration

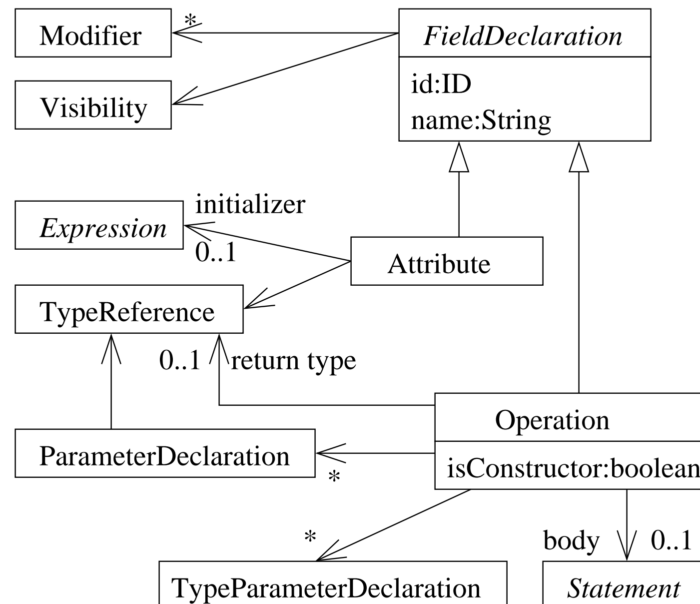


Abbildung 3.37: Deklaration von Feldern im GAST

Feldzugriffe werden im GAST über das Element *FieldAccess* (siehe Abbildung 3.38) abgewickelt. Dieses Element bestimmt das gewünschte Feld mit Hilfe der ID Feldes. Der *Qualifier* gibt an, in welcher Art und Weise der Zugriff erfolgt. Der Qualifier wird durch enthaltene Elemente näher spezifiziert:

- Der Zugriff auf Felder über Variablen, Rückgaben von Funktionsaufrufen oder Ähnliches wird über die entsprechende *Expression* ermöglicht.
- Die aktuelle Instanz einer Klasse oder Aufzählung wird durch den *this*-Qualifier erreicht. Dieser funktioniert analog zu Java oder dem *self* aus Python.
- Soll auf Felder einer der Superklassen zugegriffen werden, so wird diese Superklasse durch ein *TypeReference*-Element angegeben. Dieses Verfahren ist aufgrund der Mehrfachvererbung notwendig und ist an C++ angelehnt.
- Statische Felder werden über ein *TypeReference*-Element auf den Typ referenziert, in dem das statische Feld zugreifbar ist.

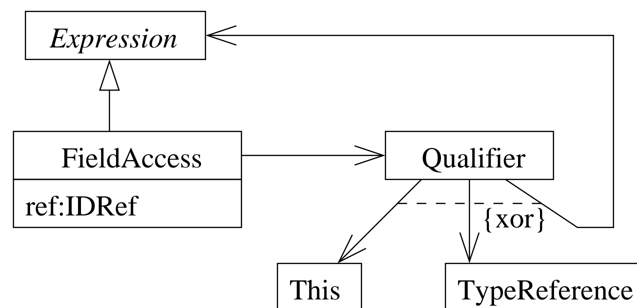


Abbildung 3.38: Zugriffe auf Felder im GAST

3.13.3 Aufzählungsdeklaration

Aufzählungen (Abbildung 3.39) enthalten als öffentlich sichtbare Elemente Konstanten. Diese werden durch das Element *ConstantDeclaration* repräsentiert und haben optional einen oder mehrere Parameter. Es handelt sich um Aufrufe der ebenfalls in der Aufzählung deklarierten Konstruktoren. Zusätzlich zu den genannten Konstanten und Konstruktoren können auch Attribute und Operationen deklariert werden.

Für den Zugriff auf diese deklarierten Konstanten werden *FieldAccess* Elemente verwendet, die auf die ID der Konstante verweisen.

Sollen klassische C Aufzählungen dargestellt werden, müssen diese simuliert werden. Siehe dazu Abschnitt 4.5.

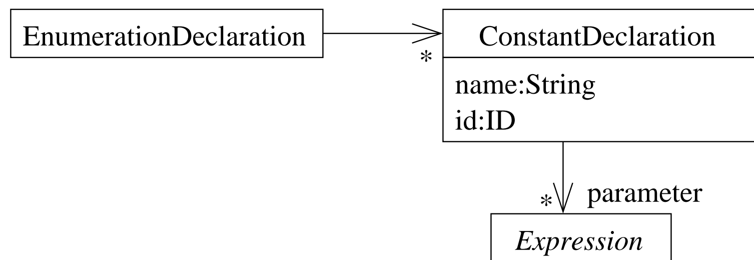


Abbildung 3.39: Details zur Aufzählungsdeklaration im GAST

Beispielsweise ist folgende Aufzählung für Farben möglich:

```
enumeration Farben
  // weitere Farben bei Bedarf einfuegen
  Schwarz(0,0,0),
  Rot(255,0,0),
  Gold(255,215,0);

  attribute Integer red;
  attribute Integer green;
  attribute Integer blue;

  operation constructor(Integer r,
      Integer g, Integer b):Farben
    red = r;
    green = g;
    blue = b;
  end

  operation grauWert():Float
    return 0,299 * red + 0,587 * green
      + 0,114 * blue;
  end
end
```

Listing 3.27: Beispiel für eine Aufzählung im GAST

Die Enumeration beinhaltet drei Konstanten (Schwarz, Rot, Gold) sowie drei Attribute für die drei Grundfarben. Der Konstruktor setzt die übergebenen Werte in die Attribute ein. Die Methode *grauWert* liefert eine reelle Zahl zurück, die den Grauwert der entsprechenden Farbe repräsentiert.

3.13.4 Deklaration komplexer Typen

Das abstrakte Element *ComplexTypDeclaration* (Abb. 3.40) erweitert die Möglichkeiten der Typdeklarationen um die Generizität. Diese kann, muss aber nicht verwendet werden. Typparameter können bei der Deklaration auf zwei Arten deklariert werden:

- Parameter ohne Typreferenz: Definiert einen Platzhalter für einen Typ. Bei einer Ausprägung wird dieser Typ an den entsprechenden Stellen der Typdeklaration substituiert. Das *Boundary*-Element belegt die ausprägenden Typen mit Einschränkungen, um bestimmte Schnittstellen zu erzwingen. Werden keine solche Beschränkungen benötigt, enthält das *Boundary*-Element keine Kindelemente.

- Parameter mit Typreferenz: Definiert einen Platzhalter für einen Ausdruck des referenzierten Typs. Bei einer Ausprägung wird der Wert des Ausdrucks dann an den entsprechenden Stellen der Typdeklaration eingesetzt. Aufgrund der Ersetzung zum Zeitpunkt des Compilierens muss der Wert des Ausdrucks beim Übersetzen feststehen.

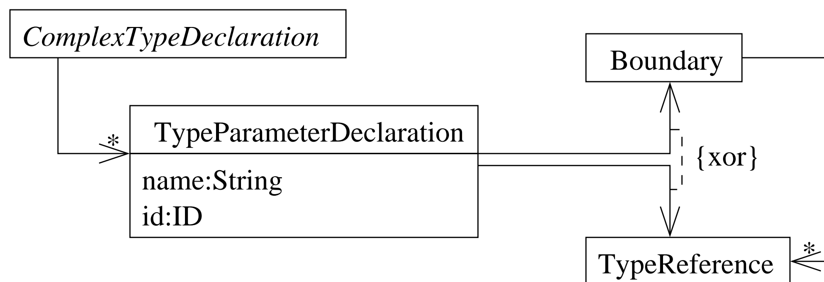


Abbildung 3.40: Details zur Deklaration komplexer Typen im GAST

Wird ein generischer Typ ausgeprägt, so werden die Typparameter gebunden. Dies geschieht durch so genannte *TypParameterBinding*-Elemente an entsprechenden *TypeReference*-Elementen. Diese Elemente beinhalten einen Verweis auf den ersetzten Typparameter. Wird ein Typparameter aus einer Deklaration mit Typreferenz ersetzt, dann muss ein zur Übersetzungszeit evaluierbarer Ausdruck angegeben sein. Ansonsten muss eine Typreferenz angegeben sein.

Um diese beiden GAST-Elemente zu erklären, werden folgende Beispiele eingeführt: Das folgende Java-Fragment `<T>` wird bei der Umsetzung in den GAST in das Objektdiagramm 3.41 umgewandelt. Das Java-Fragment `<T extends A>` wird bei der Umsetzung in den GAST in das Objektdiagramm 3.42 umgewandelt. Das Fragment `<int a>` wird bei der Umsetzung in den GAST in das Objektdiagramm 3.43 umgewandelt.

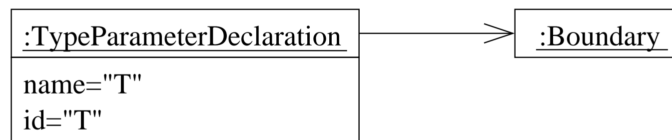


Abbildung 3.41: Objektdiagramm zur Umsetzung von <T>

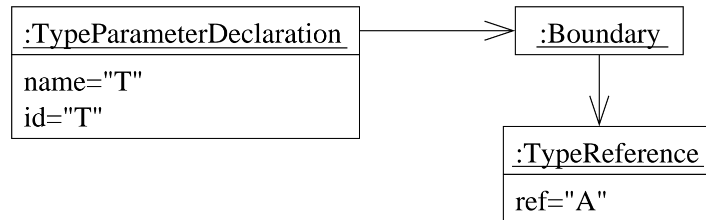


Abbildung 3.42: Objektdiagramm zur Umsetzung von <T extends A>

Ausprägungen werden durch das Element *TypeParameterBinding* repräsentiert. Die ersten beiden Beispiele von oben (Abbildung 3.41 und Abbildung 3.42) werden durch ein Fragment wie `<SomeType>` ausgeprägt. Diese Ausprägung wird im Objektdiagramm 3.44 repräsentiert. Diese Abbildung zeigt nur die Ausprägung des *TypeParameterDeclaration*-Elements aus Abbildung 3.42. Das dritte Beispiel von oben (Abbildung 3.43) wird durch ein Fragment wie `<1024>` ausgeprägt. Diese Ausprägung wird im Objektdiagramm 3.45 repräsentiert.

Ein weiteres Beispiel wird im Abschnitt über Klassendeklarationen in Listing 3.29 gegeben.

3.13.5 Schnittstellendeklaration

Schnittstellen können als Felder nur abstrakte Operationen und initialisierte Attribute beinhalten. Die Operationen sind deshalb als *abstract* zu markieren und mit leerem Rumpf zu notieren. Konstruktoren sind nicht möglich, generische Schnittstellen hingegen sind vorgesehen.

Schnittstellendeklarationen können zusätzlich zu den Möglichkeiten der Typdeklaration bestehende Schnittstellen erweitern (siehe dazu Abbildung 3.36). Dazu existieren beliebig viele Referenzen auf andere Schnittstellen, die ausdrücken, welche Schnittstellen erweitert werden.

Folgendes Codefragment deklariert eine generische Schnittstelle, die die Schnittstelle “CurrencyConverter” erweitert:

```
1 interface VarCurrencyConverter<Float ratio>
2     extends CurrencyConverter
3   attribute Float exchangeRatio = ratio;
```

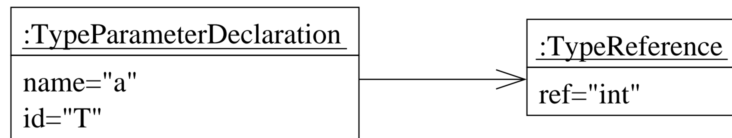


Abbildung 3.43: Objektdiagramm zur Umsetzung von <int a>

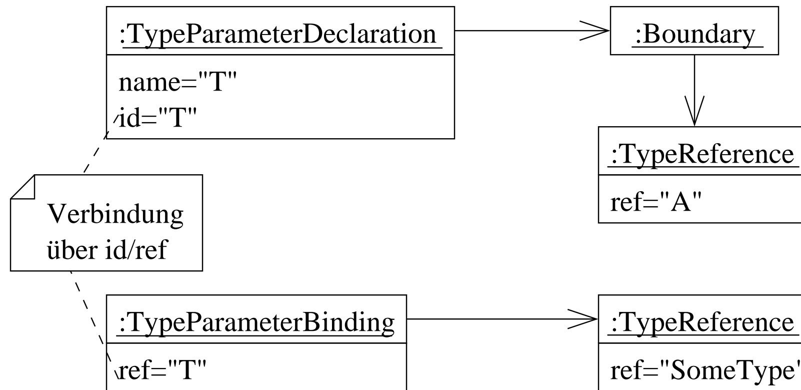


Abbildung 3.44: Objektdiagramm zur Umsetzung einer Ausprägung mit <SomeType>

```
4     abstract operation exchangeInto(Float input):Float;
5     abstract operation exchangeFrom(Float input):Float;
6 end
```

Listing 3.28: Beispiel für eine generische Schnittstelle im GAST

3.13.6 Klassendeklaration

Klassendeklarationen können Schnittstellen implementieren und Klassen erweitern. Dies wird durch die Angabe von Typreferenzen jeweils beliebiger Multiplizität mit den Rollen "superclass" und "implementing interface" dargestellt (siehe Abbildung 3.36).

Instantiierbare Klassen müssen immer mindestens einen Konstruktor besitzen. Es wird kein impliziter Standardkonstruktor wie in einigen Sprachen angelegt. Dieser und weitere Konstruktoren können, wie in Abschnitt 3.13.2 über Felddeklarationen angegeben, deklariert werden.

Die Superkonstruktoren der Superklassen müssen in den Konstruktoren explizit aufgerufen werden. Dadurch gewinnt man erhöhte Flexibilität im Vergleich zu impliziten Aufrufen, da die Aufrufreihenfolge eine sprachspezifische Angelegenheit ist. In Abbildung 3.46 ist der Aufruf eines Superkonstruktors der Superklasse "A" dargestellt, der GAST-Code ist `A.constructor();`.

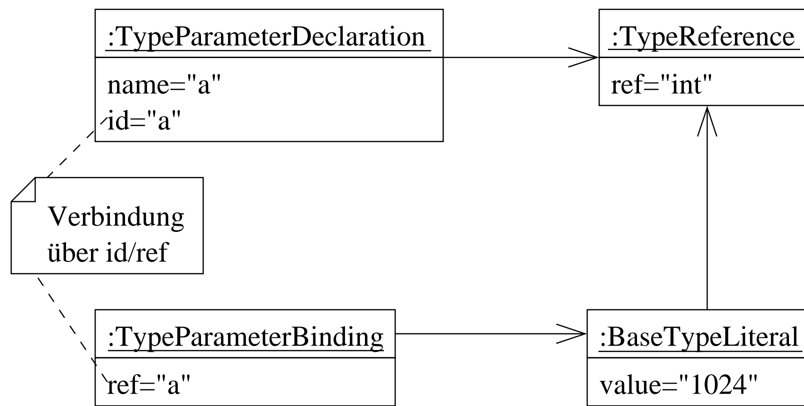


Abbildung 3.45: Objektdiagramm zur Umsetzung einer Ausprägung mit <1024>

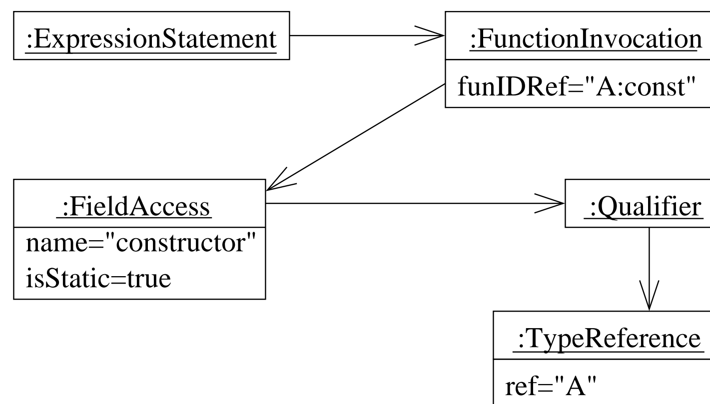


Abbildung 3.46: Details zum Aufruf eines Superkonstruktors

Im GAST-Code werden Superklassen durch das Schlüsselwort **extends** gefolgt von einer kkommasepariereteListe von Typreferenzen auf Klassen, implementierte Schnittstellen werden durch das Schlüsselwort **implements** gefolgt von einer kkommasepariereteListe von Typreferenzen auf Schnittstellen dargestellt.

Zusätzlich werden bei Klassendeklarationen (siehe Abbildung 3.47) Informationen über die Art der Klassendeklaration durch *Modifier*-Elemente ausgedrückt. Diese sind im Einzelnen:

- *abstract* bedeutet, ddasseine Instantiierung der Klasse nicht möglich ist, da mindestens eine Operation der Klasse ebenfalls abstrakt markiert worden ist. Eine gleichzeitige Verwendung von *frozen* ist nicht möglich, da sonst eine Implementierung der abstrakten Operationen verhindert würde.

- *frozen* bedeutet, ddassInstanzen der Klasse möglich sind, aber es ist keine Erweiterung der Klasse durch Verwendung als Superklasse bei neuen

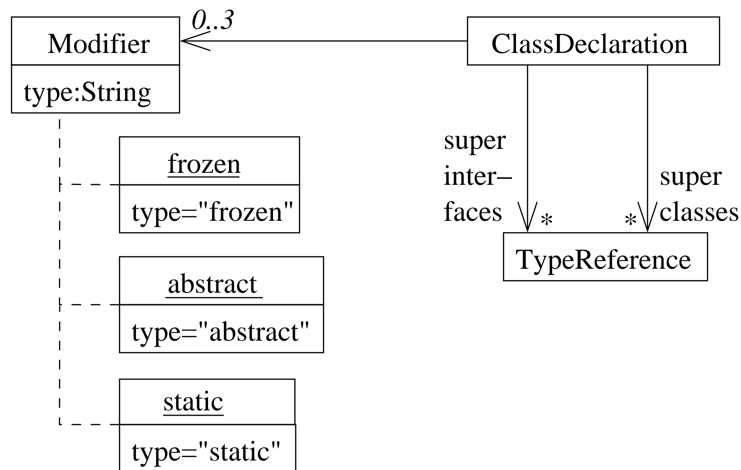


Abbildung 3.47: Details zur Deklaration von Klassen im GAST

Typdeklarationen möglich. Wie bei *abstract* bereits erwähnt, ist eine Kombination mit selbigem Modifier nicht möglich.

- *static* ist als Modifier nur bei inneren Klassen und Felddeklarationen möglich. Statische Elemente einer Typdeklaration verlieren die Bindung an ein Objekt und stehen so auch direkt über den Typ zur Verfügung. Das modifizierte Element wird dann als Klassenattribut oder Klassenmethode bezeichnet.

Die Repräsentation der Modifier als UML wird in Abbildung 3.48 dargestellt.

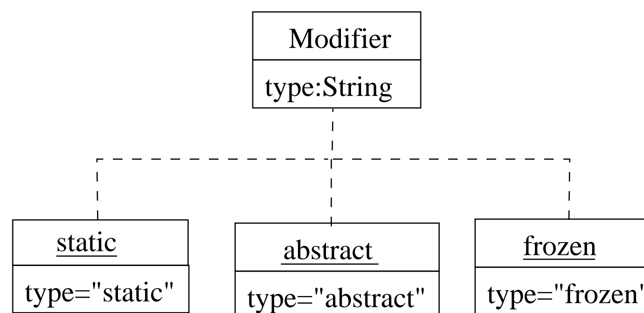


Abbildung 3.48: Details zu den Modifiern im GAST

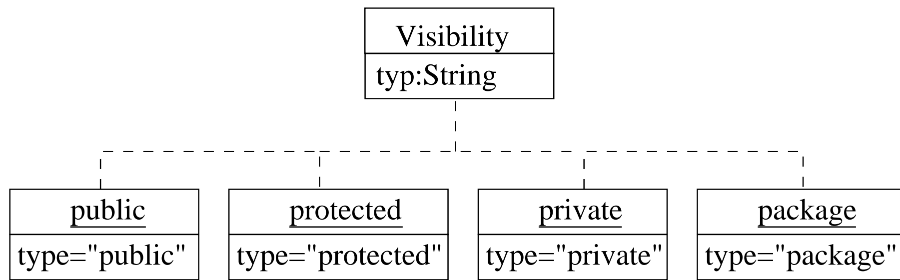


Abbildung 3.49: Details zu Sichtbarkeiten im GAST

Analog zur UML existieren vier Typen der Sichtbarkeit. Diese sind im Einzelnen:

- *public*
  Auf Elemente dieser Sichtbarkeit ist ohne Beschränkungen zugreifbar. Im GAST sind Schnittstellendeklarationen immer *public*, solange sie nicht in einer Klasse deklariert werden. Dann darf das komplette Spektrum an Sichtbarkeiten verwendet werden.

- *protected*
  Zugriffe auf Elemente dieser Sichtbarkeit sind nur von der deklarierenden Klasse und erweiternden Unterklasse möglich. Dies wird beispielsweise für Attribute und Operationen verwendet, die zwar nicht öffentlich zugreifbar sein, dennoch für abgeleitete Klassen zu Verfügung stehen sollen.

- *private*
  Auf Elemente dieser Sichtbarkeit kann nur innerhalb der deklarierenden Klasse zugegriffen werden. Zugriffe aus inneren Klassen sind möglich.

- *package*
  Mit dieser Sichtbarkeit werden Elemente ausgestattet, die für alle Klassen eines Pakets inklusive der umschließenden Pakete direkt zugreifbar sein sollen. Ein Zugriff durch Klassen außerhalb des Pakets ist nicht möglich.

Folgendes Codefragment deklariert eine öffentlich sichtbare Klasse **Puffer**. Diese besitzt zwei Typparameter: einer gibt den Typ der gepufferten Daten an, der andere Parameter die Größe des Puffers. Ein privates Attribut speichert die Daten in einem Array ab.

```
1 public class Puffer<T,GanzeZahl size>
2   private attribute Array(T,size) array;
3   // Weiterer Code
4 end
```

Listing 3.29: Beispiel für eine generische Klasse im GAST

3.13.7 Erzeugen von Objekten

Instanzen von Klassen oder Objekte werden durch Aufruf eines Konstruktors einer Klasse erzeugt. Konstruktoren liefern immer eine Instanz der deklarierenden Klasse zurück. Sie sind als statische Operationen direkt aufrufbar und jede Klasse besitzt mindestens einen Konstruktor.

Als Darstellung im GAST ist somit kein spezielles Element notwendig, um die Erzeugung eines Objektes zu repräsentieren. Es wird auf Variablenzugriffe, binäre Ausdrücke, Feldzugriffe und Funktionsaufrufe zurückgegriffen.

3.14 Sprachspezifische Fragmente

Sprachspezifische Fragmente wurden bereits im Kapitel 1.6 definiert. Hier soll nun beschrieben werden, wie diese im GAST repräsentiert werden können.

Durch den wohl üblichen Weg der Erstellung einer GAST-Instanz über einen bereits existierenden SAST liegen die sprachspezifischen Fragmente bereits in Form eines XML Dokuments vor.

Dadurch wird eine Integration der Fragmente im GAST erleichtert. In Abbildung 3.50 wird die Einbindung der SAST Struktur über ein xs:any im XML Schema beschrieben. Zur Umsetzung mit xs:any siehe [28].

Neben den Informationen des SASTs kann ein sprachspezifisches Fragment auch noch ein *Documentation*-Element enthalten. Dieses ist optional und soll, wenn möglich und sinnvoll, Verweise auf Onlineresourcen beinhalten. Dies können für Klassen und Methoden Verweise auf die API der entsprechenden Sprache sein oder sie können auf Referenzhandbücher verweisen.

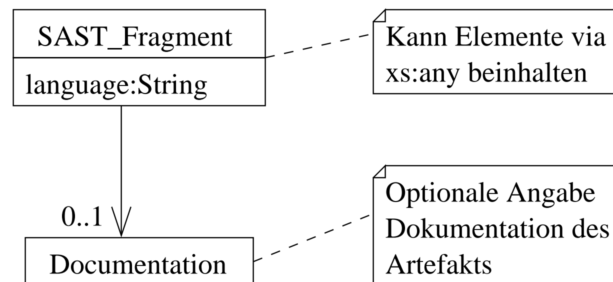


Abbildung 3.50: Sprachspezifische Fragmente im GAST (I)

Diese Form der Darstellung sprachspezifischer Fragmente deckt leider nicht den kompletten Bedarf ab. Beispielsweise könnte ein Operator eines komplexen Ausdrucks ein sprachspezifisches Fragment darstellen. Den kompletten Ausdruck als sprachspezifisch zu bezeichnen wäre falsch, da die enthaltenen Teilausdrücke durchaus darstellbar sein können. Deshalb wurde entschieden, auch in Attributen, durch ein Präfix markiert, sprachspezifische Fragmente einbetten zu können.

Um diese sprachspezifischen Fragmente auch mit XSLT auswerten zu können wurde eine sehr einfache Struktur gewählt (siehe Abbildung 3.51).

Abbildung 3.51: Sprachspezifische Fragmente im GAST (II)

Dabei stellt "Sprache" eine Folge beliebiger Zeichen mit Ausnahme des ":" dar und gibt die Herkunftssprache des Fragmentes an. "Element" eine unbeschränkte Folge beliebiger Zeichen, die das Originalfragment darstellt.

Kapitel 4

Transformationsmöglichkeiten zur Ersetzung nicht unterstützter Sprachelemente

Nicht jede reale Programmiersprache wird alle Sprachelemente des GASTs unterstützen und ebenso wird der GAST nicht alle Sprachelemente aller Sprachen unterstützen können. Deshalb sollen hier einige Transformationen beschrieben werden, die bei der Lösung dieser Probleme relevant sein können.

Hierbei unterscheiden wir drei Arten von Transformationen:

- $SAST \mapsto GAST$: Diese Transformationen ersetzen Elemente einer Quellsprache, die im GAST nicht direkt darstellbar sind.
- $GAST \mapsto GAST$: Diese Transformationen führen auf dem GAST optionale Umformungen durch.
- $GAST \mapsto SAST$: Diese Transformationen setzen den GAST in einen SAST der Zielsprache um. Diese werden in diesem Kapitel nicht behandelt, da sie zu spezifisch für einzelne Sprachen sind.

4.1 Ersetzen der Inkrement- und Dekrementoperatoren

Die Inkrement- und Dekrementoperatoren können jeweils in zwei Ausprägungen auftreten, jeweils als Prä- und Postoperator.

Dies führt zu der Entscheidung, in wie weit diese Operatoren in die Definition des GASTs aufgenommen werden.

Aufgrund der Tatsache, dass einige Sprachen diese Operatoren nur teilweise oder gar nicht unterstützen, sind sie leicht ersetzbar. Dazu sind nur Elemente des GASTs nötig, die bereits definiert wurden.

Die Ersetzungen nutzen aus, dass Zuweisungen in der GAST-Definition als Ausdrücke gelten. Diese liefern den Wert der rechten Seite der Zusweisung als Wert des Ausdrucks zurück.

Zunächst werden die beiden Präfixoperatoren behandelt. Diese Operatoren berechnen zuerst dann den neuen Wert und liefern diesen dann zurück. Somit bieten sich folgende Ersetzungen an. Aus dem Präinkrementoperator *++i* wird *(i = i + 1)* und analog wird aus dem Prädekrementoperator $--i$ der Ausdruck *(i = i - 1)*.

Danach ersetzen wir die beiden Postoperatoren. Diese Operatoren liefern zuerst den Wert der Variablen und berechnen dann neuen Wert. Deshalb ist die Umsetzung im GAST minimal komplexer. Die Ersetzung erfolgt nach folgendem Schema. Aus den Postinkrementoperator *i++* wird *((i = i + 1) - 1)* und analog wird aus dem Postdekrementoperator $i--$ der Ausdruck *((i = i - 1) + 1)*.

Nun soll noch die Äquivalenzen der Ersetzungen gezeigt werden.

4.1.1 Äquivalenz von $++i$ und $(i = i + 1)$

Sei a eine Konstante. Dann ist die Vorbedingung $i = a$, die Nebenbedingung $j = a + 1$ ist nach der Zuweisung $j = ++i$ bzw. $j = (i = i + 1)$ zu erfüllen. Die Nachbedingung lautet $a = i + 1$.

Der Operator $++i$ erfüllt alle drei Bedingungen trivialerweise per Definition.

Der Ausdruck $(i = i + 1)$ erfüllt die Vorbedingung. Die Nebenbedingung wird aufgrund der Definition des Zuweisungsoperators erfüllt. Ebenso ist die Nachbedingung erfüllt.

Analog wird die Äquivalenz von $--i$ und $(i = i - 1)$ gezeigt.

4.1.2 Äquivalenz von $i++$ und $((i = i + 1) - 1)$

Sei a wiederum eine Konstante. Auch hier ist die Vorbedingung $i = a$, die Nebenbedingung ist diesmal $a = j$ und wieder nach der Zuweisung $j = i++$ bzw. $j = ((i = i + 1) - 1)$ zu erfüllen. Die Nachbedingung lautet wieder $a = i + 1$.

Der Operator $i++$ erfüllt diese drei Bedingungen per Definition.

Auch der Ausdruck $((i = i + 1) - 1)$ erfüllt die Vorbedingung, die Nachbedingung ist aufgrund der Definition der Zuweisung erfüllt. Die Nebenbedingung $a = j$ wird erfüllt durch $j = ((i = i + 1) - 1) \rightarrow j = ((a + 1) - 1) \rightarrow j = a$.

Analog wird die Gültigkeit der Ersetzung von $i--$ durch $((i = i - 1) + 1)$ gezeigt.

4.2 Umsetzung der elementweisen, booleschen Operatoren *Und* und *Oder*

Diese Operatoren existieren nicht in der Definition des GAST. Sie können aber einfach ersetzt werden. Auch hierbei handelt es sich um eine $SAST \mapsto GAST$-Transformation.

Da in diesen beiden Fällen jeweils beide Ausdrücke ausgewertet werden, bietet sich eine Ersetzung durch folgende Funktionen an:

```
1 Function and = function(Boolean a1, Boolean a2):Boolean
2   return (a1 && a2);
3 end
4
5 Function or = function(Boolean a1, Boolean a2):Boolean
6   return (a1 || a2);
7 end
```

Listing 4.1: Alternativen zu den elementweisen Operatoren **und** und **oder**

Durch den Funktionsaufruf werden zunächst beide Ausdrücke ausgewertet und dann mithilfe der bekannten Operatoren verknüpft.

4.3 Ersetzen von *break* und *continue* mit Zielmarken

Diese sind zwar im GAST vorgesehen, jedoch nicht in allen Zielsprachen vorhanden oder erwünscht.

Deshalb werden in diesem Abschnitt Verfahren eingeführt, die es erlauben, GAST-Instanzen automatisiert in GAST-Instanzen ohne Break- und Continue-Anweisungen zu transformieren. Es handelt sich hierbei um eine $GAST \mapsto GAST$-Transformation.

Zuerst einmal einige für die Algorithmen notwendige Definitionen.

Kompositionen (K) sind Anweisungen, die weitere Anweisungen beinhalten können, nicht jedoch Schleifen. Im GAST sind dies Blöcke, If-Anweisungen, Switch-Anweisungen und die Monitor-Anweisung.

Schleifen (S) sind Anweisungen, die eine Form der Wiederholung beinhalten. Im GAST sind das Foreach-, For- und bedingungsgesteuerte Schleifen.

Grundblöcke (G) sind beliebige Anweisungen, die eine abgeschlossene Einheit bilden. Das bedeutet, sie umfassen nicht die gerade betrachtete Break- oder Continue-Anweisung.

Unterbrechungen (U) sind Anweisungen, die eine Abweichung vom normalen Kontrollfluss einleiten. Im GAST sind dies Break- oder Continue-Anweisungen. Für die beschriebenen Verfahren werden noch Anweisungen, die Unterbrechungen als Unteranweisungen beinhalten, hinzugenommen.

4.3.1 Verfahren für Break-Anweisungen

Das Verfahren zum Ersetzen einer Break-Anweisung funktioniert wie folgt:

1. Durchsuche den Code nach Break-Anweisungen. Sei m die Marke der Anweisung.

2. Prüfe, ob Marke m an einem Element auf der Parent-Achse der Break-Anweisung steht. Beschränkt ist dies durch die Deklaration der umfassenden Funktion/Methode oder Paketdeklaration. Wenn nein, fehlerhafte GAST Instanz. Ansonsten wird das Verfahren fortgesetzt.

3. Einfügen eines booleschen Flags vor der Anweisung mit Label m. Setze das Flag auf *false*.

4. Ersetze die behandelte Break-Anweisung durch eine Zuweisung des Wertes *true* an das Flag.

5. Bearbeite umfassende Elemente des GASTs ausgehend von der ersetzten Break-Anweisung bis zum Label m nach den angegebenen Regeln. Die referenzierte Bedingung ist hierbei immer $(!flag)$ bzw. eine Verknüpfung der Form $(!flag)\&\&$.

    (a) In Blöcken werden alle nachfolgenden Geschwister der unterbrechenden Anweisung durch eine if-Anweisung mit der oben genannten Bedingung umgeben.

    (b) Bei bedingungsgesteuerten Schleifen wird die Bedingung um eine Verknüpfung der oben genannten Form erweitert. Aus exemplarisch `while(c1) stmt` wird so `while((!flag)&&c1) stmt`.

    (c) Bei **If**- und **Monitor**- sowie **SwitchStatements** müssen keine Änderungen durchgeführt werden.

    (d) Bei **foreach**-Schleifen wird eine *if*-Anweisung mit der oben genannten Bedingung vor dem ursprünglichen Rumpf eingefügt. Aus `foreach(..) stmt` wird somit `foreach(..) if(!flag) stmt`.

    (e) Bei *For*-Schleifen werden die folgenden zwei Anpassungen nötig:

        i. Erweitere die Bedingung der For-Schleife um $(!flag)\&\&$, um eine weitere Iteration zu verhindern.

ii. Erweitere die Ausdrücke im Calculation-Abschnitt einzeln um folgendes Fragment $(!flag)\&\&$, so dadassiese nicht mehr nach dem Rumpf der Schleife ausgewertet werden.

Aus `for(init;cond;update) stmt` wird durch diese Transformationen `for(init; (!flagg) && cond;(!flagg) && update) stmt`.

Sind alle break- und continue-Anweisungen ersetzt, so können die entsprechenden Labels gelöscht werden.

4.3.2 Verfahren für Continue-Anweisungen

Das Verfahren zum Ersetzen einer Continue-Anweisung funktioniert wie folgt:

1. Durchsuche den Code nach Continue-Anweisungen. Sei m die Marke der Anweisung.

2. Prüfe, ob Marke m an einem Element auf der Parent-Achse der Continue-Anweisung steht. Beschränkt ist dies durch die Deklaration der umfassenden Funktion/Methode oder Paketdeklaration. Wenn nein, fehlerhafte GAST Instanz. Ansonsten wird das Verfahren fortgesetzt.

3. Einfügen eines booleschen Flags vor der Anweisung mit Label m. Setze das Flag auf *false*.

4. Ersetze die behandelte Continue-Anweisung durch eine Zuweisung des Wertes *true* an das Flag.

5. Bearbeite umfassende Elemente des GASTs ausgehend von der ersetzten Continue-Anweisung bis zum Label m nach den angegebenen Regeln. Die referenzierte Bedingung ist hierbei immer $(!flag)$ bzw. eine Verknüpfung der Form $(!flag)\&\&$.

   In Blöcken werden alle nachfolgenden Geschwister der unterbrechenden Anweisung durch eine if-Anweisung mit der oben genannten Bedingung umgeben. Bei allen weiteren Elementen werden keine Änderungen nötig.

Sind alle break- und continue-Anweisungen ersetzt, so können die entsprechenden Labels gelöscht werden.

4.4 *fall-through switch-statements* in GAST Code umsetzen

Soll eine Switch-Case-Anweisung mit *fall-through* Semantik im GAST in eine Switch-Case-Anweisung ohne *fall-through* Semantik umgesetzt werden, so muss

dieser Unterschied durch eine Anpassung der Struktur der Switch-Case-Anweisung kompensiert werden. Es handelt sich hierbei um eine Transformation $GAST \mapsto GAST$.

Hier wird nun ein Verfahren erläutert, welches diese Anpassung der Struktur automatisch vornehmen kann.

1. Suche Switch-Case-Anweisung S
2. Durchlaufe alle Case-Clauses C der Anweisung S in aufsteigender Reihenfolge und wende folgende Schritte auf C an:
   (a) Bestimme das von C aus nächste Case C_B mit unbedingter Break-Anweisung am Ende des Anweisungsblocks. Wird kein solches Break gefunden, so wird bis zum Ende der Switch-Case-Anweisung gegangen.
   (b) Übernimm die kompletten Anweisungsblöcke aller Case-Clauses zwischen C und C_B (jeweils einschließlich).

Als Ergebnis erhält man eine semantisch äquivalente Version der Switch-Case-Anweisung. Diese beinhaltet noch eventuelle Break-Anweisungen in den Case-Teilen; Diese können aber bei Bedarf mit Hilfe einer $GAST \mapsto GAST$-Transformation entfernt werden.

4.5 Klassische C Aufzählungen

Klassische Enumerationen in C oder C++ sind Aufzählungen, die nur ganzzahlige Werte annehmen. Zum Beispiel:

```
1 enum FooSize {
2           SMALL = 10,
3           NOT_SO_SMALL,
4           MEDIUM = 100,
5           LARGE = 1000
6 };
```

Listing 4.2: klassische C Enum

Die Semantik dieser Aufzählungen ist im Prinzip eine Menge von konstanten ganzzahligen Werten. Sie können an Integervariablen per Name zugewiesen werden. Die Konstante “NOT_SO_SMALL” besitzt den Wert 11.

Exemplarisch wird diese nun in eine GAST-Enumeration umgesetzt.

```
1 enumeration FooSize
2   SMALL(10),
3   NOT_SO_SMALL(),
4   MEDIUM(100),
5   LARGE(1000);
```

```
6
7   static attribute Integer counter=0;
8   attribute Integer value;
9
10  operation constructor = function():FooSize
11    value=counter;
12    counter = counter + 1;
13  end
14
15  operation constructor = function(Integer v):FooSize
16    value = v;
17    counter = v +1;
18  end
19
20  operation getValue = function():Integer
21    return value;
22  end
23 end
```

Listing 4.3: Umsetzung einer C-Enum in GAST

4.6 Umsetzung von Casts in GAST

Im GAST wurde, analog zu Zuweisungen, auch für Casts kein explizites Element in die Definition eingefügt.

Casts werden in nahezu allen Sprachen syntaktisch analog aufgebaut. Normalerweise wird der Wert eines Ausdrucks in einen angegebenen Typ umgewandelt. Java Casts werden in Listing 4.4 exemplarisch gezeigt.

```
1         String s = "hallo";
2         Object o = (Object) s;          //upcast
3         String s1 = (String) o;         //downcast
```

Listing 4.4: Beispiele für Casts in Java

Die Lösung besteht aus einem binären Operator, der in Abschnitt 3.11.4 eingeführt wurde.

Das Objektdiagramm in Abbildung 4.1 verdeutlicht die Umsetzung der dritten Zeile aus Listing 4.4.

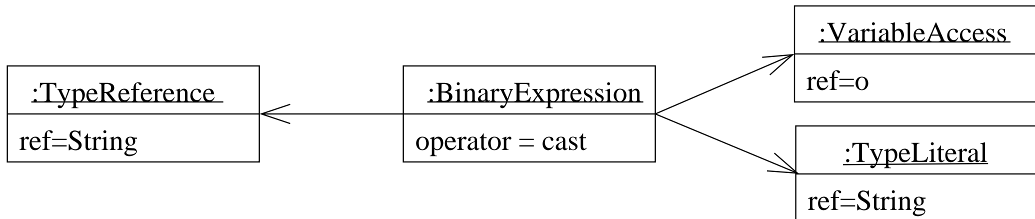


Abbildung 4.1: Umsetzung des Casts aus Listing 4.4 Zeile 3

Teil II

Praxis

Kapitel 5

JaML - Java Markup Language

5.1 Einführung

JaML ist eine Technologie, die an der Universität Würzburg entwickelt wurde und bereits mehrere interne Evolutionsstufen durchlaufen hat. JaML übersetzt Java-Quellcode in eine mit Informationen angereicherte Darstellung und legt diese in `XML`-Dateien ab. Es handelt sich bei dieser Stufe am ehesten um einen attributierten Parsebaum mit Informationen über alle Zeichen (siehe Kapitel 1.3), da der Baum neben dem kompletten Programm zusätzlich Informationen zu den Anweisungen, Ausdrücken und Typen beinhaltet.

5.2 Anforderungen an JaML

Folgende Anforderungen sollten durch die JaML Dateien erfüllt werden:

1. Direkte Wiederherstellbarkeit des Originalquelltextes aus `JaML`-Dateien. Hier muss eine Einschränkung hingenommen werden: Das Format der optional markierten Zeilenumbrüche wird zwar in den entsprechenden Tags als Attribut gespeichert, jedoch im XML nur als UNIX Variante abgelegt. Dadurch kann es zu Veränderungen am Format der Zeilenumbrüche kommen. Ebenfalls optional markiert JaML Whitespaces und fügt Angaben über die Positionen der Elemente im Quellcode ein.

2. Validierbarkeit nach einem XML-Schema.

3. Basisformat für spätere Abstraktionen. Das bedeutet, dass bereits in den JaML Dateien in den XML Tags alle relevanten Informationen enthalten sind und die Abstraktionen sich auf das Entfernen von Elementen und Transformationen der XML Bäume beschränken.

4. Unterstützung für alle Konstrukte der Sprache Java bis einschließlich Version 1.5. Das bedeutet speziell, dass generische Typen sowie Enumerationen

und Annotationen als auch die erweiterte *for*-Schleife berücksichtigt werden müssen.

5. Einfache Einsatzmöglichkeiten, sowohl in einer IDE als auch Skriptfähigkeit zur Integration in bestehende Applikationen. Eine Einsatzmöglichkeit der hier beschriebenen Darstellungen (JaML, JAST und GAST) sehe ich bei dem von unserem Lehrstuhl entwickelten Java Online Tutorial und den damit verbundenen automatisierten Tests der Software (siehe [13] in [14]).

Weniger berücksichtigt wurden hierbei Bedenken gegen eine Speicherung in XML, die in [10] vorgetragen wurden. Diese Veröffentlichung vergleicht XML als Speicherformat des ASTs mit einem proprietären binären Format im Bezug auf Dateigröße, Speicherverbrauch zum Bearbeiten, Bearbeitungsgeschwindigkeit und Interoperabilität zwischen Programmen. Die Dateigröße liegt laut Paper bei XML niedriger. Beim Speicherverbrauch und der Zeit liegt das binäre Format vorne, jedoch ist XML wieder der Gewinner bei der Interoperabilität. Das liegt in der Natur des standardisierten Formats. Da diese Diplomarbeit ein sprachunabhängiges Format als Ziel besitzt, wurde sinnvollerweise hoher Wert auf die Interoperabilität gelegt und deshalb eine Entscheidung zugunsten von XML getroffen.

5.3 Implementierung als Plug-in für Eclipse

Zur Implementierung von JaML, also im Prinzip eines Übersetzers von Java nach XML, lag es nahe, einen bereits existierenden Übersetzer zu verwenden.

Dieser musste aufgrund der Anforderungen an JaML Java 1.5 unterstützen und sollte möglichst eine aktive Entwicklergemeinde besitzen. Dies ist wünschenswert, um eventuell vorhandene Fehler in diesem essentiellen Stück der Software nicht selbst beheben zu müssen.

Die Wahl fiel aufgrund dieser Kriterien auf Eclipse und das Unterprojekt JDT, also das Java-Development-Tools Projekt (siehe [25]). Das JDT bietet neben einer eigenen Implementierung eines Java Compilers auch einen direkten Zugang zu dem generierten Syntaxbaum. Dieser Zugang wird im nächsten Abschnitt genauer betrachtet.

Die Anforderung, JaML in eine IDE zu integrieren, konnte durch das Plug-in-Konzept von Eclipse (Siehe [11]) relativ leicht umgesetzt werden. Dies war auch während der Entwicklung und in der Testphase sehr hilfreich.

5.3.1 Relevante Informationen zu Eclipse

Dieser Abschnitt bietet nur eine sehr kurze Einführung in die interne Architektur der Entwicklungsumgebung Eclipse und stellt kurz die zur Implementierung notwendigen Klassen vor.

In Eclipse ist ein Projekt auf Dateisystemebene zunächst nur eine Ansammlung von Verzeichnissen und Dateien. Über "Nature" genannte Klassen, die das Interface **org.eclipse.core.resources.IProjectNature** implementieren, werden den Projekten weitere Eigenschaften zugewiesen.

Eine dieser Eigenschaften ist beispielsweise ein individueller "Builder". Diese ermöglichen eine besondere Behandlung bestimmter Projekte und Dateien. Ein Beispiel für einen solchen Builder ist der "Java Builder", der in Verbindung mit der "Java Nature" von Eclipse den Java-Projekten automatisch zugeordnet wird.

Um Eclipse zu erweitern, existiert ein Plug-in Mechanismus. Soll ein solches Plug-In einen Beitrag zur graphischen Oberfläche liefern, so wird die Klasse **org.eclipse.ui.plugin.AbstractUIPlugin** erweitert.

Um persistente Einstellungen zu Projekten ablegen zu können, wird eine **PropertyPage** verwendet. Dazu wird die Klasse **org.eclipse.ui.dialogs.PropertyPage** erweitert. Sie bildet eine graphische Schnittstelle und erlaubt dem Benutzer das zugehörige Plug-in zu konfigurieren.

5.3.2 Details der Implementierung

Auf die Details der Implementierung wird, durch einige Sequenzdiagramme unterstützt, in diesem Abschnitt eingegangen.

Das Diagramm in Abbildung 5.1 beschreibt die vorbereitenden Schritte, die notwendig sind, um ein Eclipse Projekt mit dem **JaMLBuilder** zu bearbeiten. Der Build-Vorgang wird extern durch Aufrufen der Methode *build()* angestoßen. Dabei wird ein **FileVisitor** erzeugt und diese Instanz an das zu bearbeitende Projekt durch die Funktion *accept()* weitergereicht. Intern wendet Eclipse dann ein Besuchermuster an und die Funktion *visit()* wird für jeden Knoten des Verzeichnisbaums aufgerufen.

Der eigentliche Vorgang des Parsens und die zweite Anwendung des Besuchermusters beschreibt das Diagramm in Abbildung 5.2. Die Methode *visit()* ruft zunächst die Methode *process()* auf. Diese erzeugt dann einen **ASTParser** und einen **JaMLASTVisitor**. Der Aufruf der Methode *createAST()* erzeugt den attributierten Parsebaum und liefert die Wurzel als **ASTNode** zurück. Auf dieser Wurzel wird dann die Methode *accept()* mit dem erzeugten **JaMLASTVisitor** aufgerufen. Dadurch wird das zweite Mal das Besuchermuster angewendet und auf der **JaMLASTVisitor**-Instanz für jeden Knoten des Parsebaums eine polymorphe *visit()*-Methode aufgerufen.

Um nun `XML`-Dateien zu erhalten, wird durch den **CodeAnnotator** ein SAX Ereignisstrom erzeugt, der nacheinander durch drei Filter geleitet wird. Diese Filter erzeugen durch den **KeyWordSymbolFilter** den Markup für Symbole und Schlüsselwörter der Sprache; Der **WhitespaceNewlineFilter** fügt die Tags für Whitespaces aller Art (Leerzeichen, Tabulatoren, Zeilenumbrüche) ein. Dieser Filter kann, wie auch der folgende, über die **PropertyPage** des Plug-ins optional

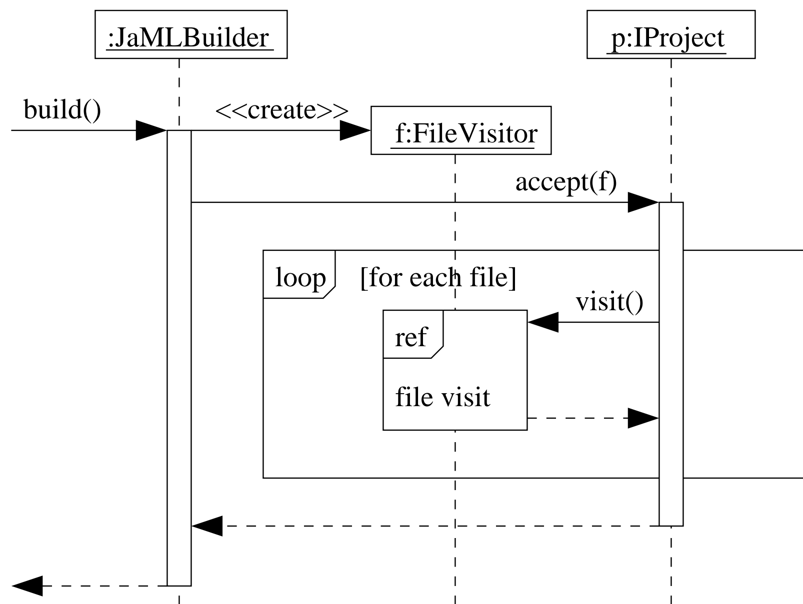


Abbildung 5.1: Sequenzdiagramm zum Buildvorgang (1) - Vorbereitungen

eingeschaltet werden. Der letzte Filter, der **PositionInformationFilter**, fügt Angaben zu den Positionen der einzelnen Elemente im Quellcode ein.

Danach wird der SAX Ereignisstrom in ein DOM Dokument transformiert, da die nun folgenden Transformatoren leichter auf einem DOM Dokument umsetzbar sind. Diese Transformatoren implementieren das Interface **ITransformer** (siehe Abb. 5.3) und befinden sich alle im Paket de.uniwue.i2.jaml.helpers.xml.transformer. Verwendet werden diese Transformatoren, um strukturelle Probleme der `XML`-Daten zu beheben.

Die Art der Probleme ist häufig etwas wie: Einfügen eines Tags, nachdem ein anderes Tag irgendwo aufgetreten ist oder dem Umhängen ganzer Unterbäume. Solche Veränderungen sind mit SAX-basierter Programmierung sehr schwer bis unmöglich zu implementieren. Deshalb wurde der Schritt hin zu einer Darstellung als DOM Baum gemacht.

Diese Probleme an der Struktur der Daten entstehen durch das Prinzip des Einfügens der Tags in den Quellcode. Beispielsweise beginnen Blöcke im AST von Eclipse am Anfang der ersten Anweisung und enden am Ende der letzten Anweisung. Also stünden ohne Korrektur die <Block>-Tags nicht um die geschweiften Klammern, sondern nur um die Anweisungen herum. Bei einem leeren Block wäre für den AST somit kein Block vorhanden. Auch dies wird durch einen entsprechenden Transformator behoben.

Nach diesen Transformationen kann der DOM Baum nun als fertige `JaML`-Datei auf die Festplatte geschrieben werden.

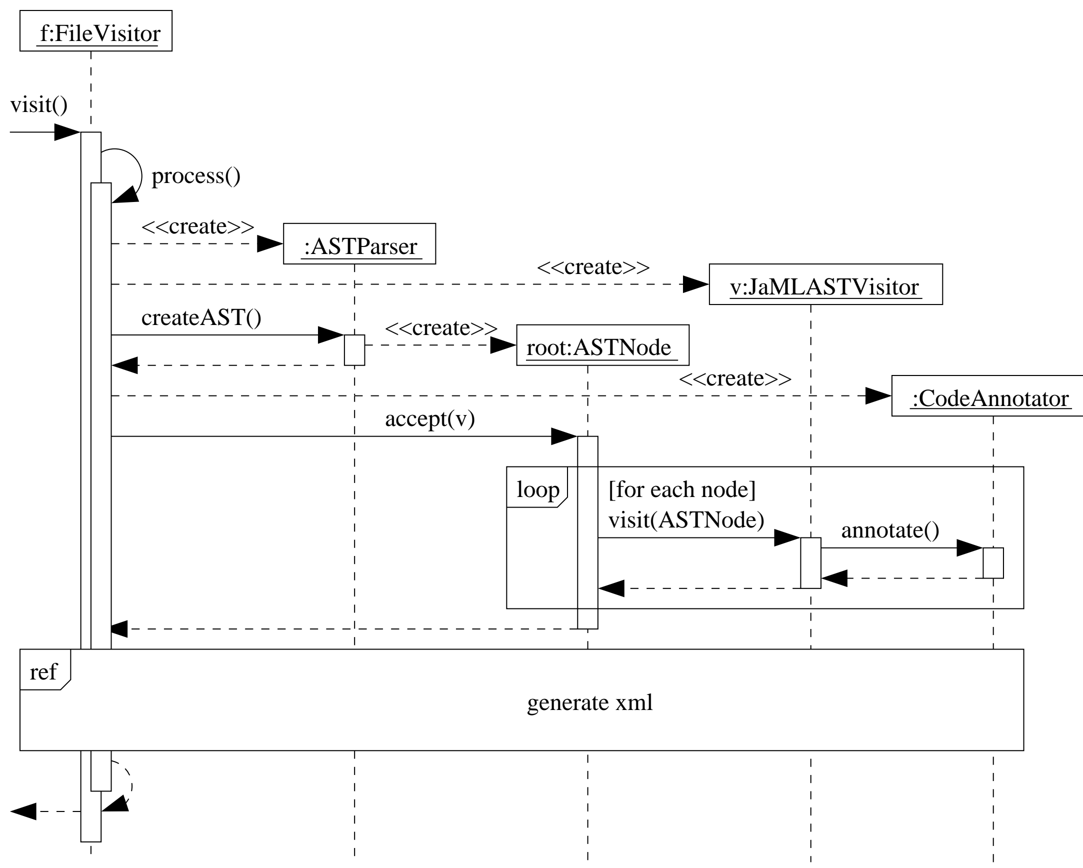


Abbildung 5.2: Sequenzdiagramm zum Buildvorgang (2) - Parsevorgang

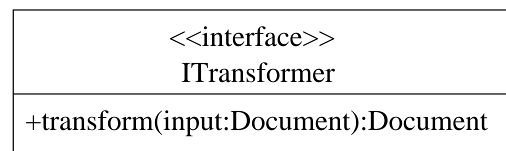


Abbildung 5.3: Das Interface **ITransformer**

5.3.3 Probleme bei der Implementierung

Bei der Implementierung von JaML traten natürlich verschiedene Probleme auf. Im Folgenden werden einige dieser Probleme analysiert und entsprechende Lösungen dargestellt.

Fehler in den Syntaxbäumen des verwendeten Compilers

Einen Fehler verursachte der Parser in Verbindung mit Codefragmenten, die den **instanceof** Operator mit generischen Typen verwendeten (Beispielsweise **r instanceof List<Integer>**). Dabei wurde die Länge des **InstanceofExpression** Knoten durch Eclipse falsch angeben, und zwar wurden die generischen Anteile (hier **<Integer >**) nicht mit in die Länge eingerechnet. Dieser Bug wurde im Bugzilla des Eclipse Projektes gemeldet und in Version 3.2 behoben. Siehe dazu [19].

Ein weiterer Fehler betraf die Länge von Variablendeklarationen bei vollqualifizierten Klassennamen in Verbindung mit Arrays. Dabei wurden die eckigen Klammern nicht mitgerechnet und somit war auch hier die Länge im AST falsch. Auch dieser Bug wurde gemeldet und ist seit Version 3.2 behoben. Siehe dazu [20].

Ebenfalls gemeldet und behoben wurde ein weiterer Fehler [21], der ebenfalls eine fehlerhafte Länge in den entsprechenden Knoten im AST zur Folge hatte. Konkret bezog sich der Fehler auf geklammerte Ausdrücke im Initialisierungsteil von For-Schleifen.

Diese Fehler waren den Eclipse Entwicklern bisher nicht aufgefallen, da sich die Fehler nur auf Längenangaben im AST bezogen und hatten daher keine Auswirkungen auf die eigentliche Funktionalität des Compilers. Der nächste Fehler hingegen kann als ernsthafter Fehler angesehen werden, da Eclipse auch Variablen primitiven Typs (int, char, ...) auf der linken Seite von **instanceof**-Operatoren zuließ. Folgendes Codefragment wurde von Eclipse ohne Fehlermeldung akzeptiert:

```
1 int a = 100;
2 boolean c = a instanceof Object;
```

Listing 5.1: Beispiel zum instanceof Fehler

Dieser Fehler wurde ebenfalls im Bugzilla (siehe [18]) eingetragen und in Version 3.2 behoben.

Alle diese Fehler wurden im Verlauf der Diplomarbeit entdeckt, gemeldet und auch durch die Entwickler behoben.

Der vollqualifizierte Klassenname bei lokalen und anonymen Klassen

Laut der Java Language Specification [15] besitzen lokale und anonyme Klassen keinen vollqualifizierten Klassennamen.

Eine lokale Klasse ist eine geschachtelte Klasse, die kein Element irgendeiner Klasse ist und einen Namen besitzt (siehe [15] Kapitel 14.3). Ein Beispiel für eine lokale Klasse sieht man im folgenden Listing.

```
1 void b() {
2     class LocalClass {
3         ...
4     }
5 }
```

Listing 5.2: Eine lokale Klasse

Eine anonyme Klasse ist eine Klasse, die ohne Angabe eines Namens definiert wurde. Dies wird häufig in der Entwicklung grafischer Benutzeroberflächen eingesetzt, um **ActionListener** und ähnliche Funktionalitäten zu implementieren. Eine Deklaration einer anonymen Klasse sieht man im nächsten Listing.

```
1 return new Iterator<Object>() {
2     //Hier ist der Rumpf der anonymen Klasse
3     ...
4 };
```

Listing 5.3: Eine anonyme Klasse

Problematisch ist hierbei das Fehlen eines vollqualifizierten Klassennamens bei den lokalen Klassen. Der vollqualifizierte Klassenname wird bei der Transformation von JAST nach Java verwendet, um eindeutige Referenzierungen ohne **import**-Anweisungen zu erhalten. In besonderen Fällen kann es nun jedoch nötig sein, von lokalen Klassen ausgehend, eine Hierarchie von Klassennamen aufzubauen. Als Beispiel dient folgender Quellcode:

```
1 public class X11 {
2     void c() {
3         class LocalClass {    // die lokale Klasse
4             class InnerClass {    //die innere Klasse
5                 ...
6             }
7             ...
8         }
9         ...
10        InnerClass ic = new InnerClass();
11    }
12 }
```

Listing 5.4: Eine lokale Klasse mit innerer Klasse

Bei der Transformation von JAST nach Java wird die Klasse *InnerClass* qualifiziert mit **LocalClass**. Man erhält somit folgende neue Zeile 10:

```
10 LocalClass.InnerClass ic = new LocalClass.InnerClass();
```

Listing 5.5: qualifizierte Angabe der inneren Klasse

Dieser teilqualifizierte Klassenname wird wie folgt bestimmt: Jede Klasse besitzt in Java einen eindeutigen "binary name"(siehe [15] Kapitel 13.1). Dieser besteht für eine lokale Klasse aus dem binären Namen der Klasse, in der die Methode definiert ist, gefolgt von dem Zeichen $ und einer nichtleeren Folge von Ziffern sowie dem Namen der lokalen Klasse. In unserem Beispiel kann der binäre Name der Klasse **LocalClass** einfach die Zeichenkette **X11$1LocalClass** und der binäre Name der Klasse **InnerClass** die Zeichenkette **X11$1LocalClass$InnerClass** sein. Dieser binäre Name wird nun verwendet, um die gewünschte Form des teilqualifizierten Namens zu erhalten. Zuerst wird der binäre Name an den $ Zeichen, die von mindestens einer Ziffer gefolgt werden, in Blöcke aufgeteilt. Der Name der lokalen Klasse wird nun in diesen Blöcken gesucht. Dann wird aus diesem Block der teilqualifizierte Name der lokalen Klasse gewonnen. In unserem Beispiel erhalten wir somit als teilqualifizierten Namen der Klasse **InnerClass** die Zeichenkette **LocalClass.InnerClass**.

Diese teilqualifizierten Namen werden nicht in JaML direkt benötigt, sondern erst bei der Transformation von JAST (siehe Kapitel 6.4) zurück nach Java.

Das Attribut *isLocal* bei Feldzugriffen

Bei Zugriffen auf Felder von Klassen, öffentliche Konstanten aus Interfaces oder Enumerationen ist es wichtig zu wissen, ob es sich um einen lokalen Zugriff handelt. Diese Information wird benötigt, wenn das aus JaML abgeleitete JAST wieder in Java umgesetzt werden soll. Bei lokalen Zugriffen darf das Feld nicht qualifiziert werden, bei nicht-lokalen Zugriffen muss qualifiziert werden um korrekten Java Code zu erhalten.

Zwei Beispiele für lokale und nicht-lokale Zugriffe zeigen die beiden folgenden Listings.

```
1 class FieldAccess {
2     private int field;
3
4     void b() {
5         //hier ist der lokale Zugriff auf das Feld field
6         field = 5;
7     }
8 }
```

Listing 5.6: Ein lokaler Feldzugriff

```
1  enum Color {
2      red, blue, green;
3  }
4
5  class FieldAccess {
6      void b() {
7          //hier ist der nicht lokale Feldzugriff
8          Color = Color.red;
9      }
10 }
```

Listing 5.7: Ein nicht-lokaler Feldzugriff

Auch dieses Problem trat erst beim Transformieren von JAST nach Java auf, da nicht klar war, welche Zugriffe bei der Transformation qualifiziert werden müssen.

5.4 Die Implementierung der Kommandozeilenversion

Neben der Version als Eclipse-Plug-in existiert auch die Möglichkeit, JaML auf der Kommandozeile und somit auch in Skripten, zu verwenden. Voraussetzung hierzu ist jedoch eine lauffähige Installation der Eclipse Entwicklungsumgebung. Weiteres hierzu findet man in Kapitel 8.

Die Kommandozeilenversion des Plug-ins wird über den "extension point" `org.eclipse.core.runtime.applications` eingebunden, der eine Implementierung des Interfaces **IPlatformRunnable** erwartet. Diese Implementierung ist hier die Klasse **de.uniwue.i2.jaml.CliJaML**, deren Methode *run* zuerst die Kommandozeilenparameter festlegt und die korrekte Verwendung prüft. Dann wird im angegebenen Workspace (siehe Abschnitt 8.4.2) ein temporäres Projekt mit Namen 'JAML' erzeugt. In diesem wird dann der Classpath mit den Werten von der Kommandozeile (Option `-cp` / `--classpath`) gefüllt und anschließend ein Link auf das Basisverzeichnis (angegeben mit der Option `-b` / `--basepath`) eingefügt. Als Ergebnis erhält man ein Eclipse Projekt mit allen notwendigen Klassen, um die `Java`-Dateien in `JaML`-Dateien zu transformieren. Anschließend wird die Transformation mithilfe der bereits beschriebenen Schritte und Klassen durchgeführt. Nach der Transformation wird das temporäre Projekt wieder aus dem Workspace entfernt. Die Ergebnisse werden dann im angegebenen Basisverzeichnis abgelegt.

5.5 Transformation von JaML nach Java

Die Rücktransformation von JaML Dateien in Java Quellcode kann sehr einfach durch Anwendung einer XML Transformation (XSLT) erreicht werden. Weiterführende Informationen zu XSLT findet man in [27], eine Einführung zum Thema in [6].

Das verwendete Stylesheet gestaltet sich sehr einfach und kurz:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        version="1.0">
    <xsl:output method="text"/>
</xsl:stylesheet>
```

Listing 5.8: Das XML Stylesheet zur Erzeugung von Java Quellcode

Dieses Stylesheet entfernt einfach alle Tags aus dem JaML Dokument, da der Quellcode des Programms komplett als Textknoten im Dokument abgelegt ist. Dies war eine Anforderung an das Format von JaML.

5.6 Test der Transformationen zwischen JAST und Java

Um die Transformationen aus diesem Abschnitt zu testen, wurde auf den Quellcode aus der Datei `src.zip` zurückgegriffen, die jeder Java Distribution beiliegt. Aufgrund der hohen Anzahl von Zeilen und des hohen Codeumfangs war dies eine ideale Testgrundlage.

Für die Tests wurden zunächst alle enthaltenen `Java`-Dateien in `JaML`-Form gebracht und anschließend zurück nach Java transformiert. Die `JaML`-Dateien wurden gegen das Schema `src/schema/JaML/JaML.xsd` validiert. Nachdem aus den `JaML`-Dateien der Originalcode bis auf die Art der Leerzeichen und Whitespaces wieder herstellbar ist, wurde das bekannte Unix Werkzeug `diff` verwendet, um dieses Kriterium zu prüfen.

Dieser Test wurde von der vorliegenden Implementierung durchlaufen und bestanden.

Kapitel 6

JAST - Ein SAST für Java

6.1 Einführung

Bei JAST handelt es sich um einen sprachspezifischen abstrakten Syntaxbaum (specific abstract syntax tree, kurz SAST) für Java. Dieser besitzt ein Format, welches sich stark an das Format von JaML anlehnt. Die Veränderungen des Formats betreffen nahezu ausschließlich den höheren Abstraktionsgrad. Die Hauptunterschiede zwischen den beiden Formaten werden im folgenden Abschnitt genau betrachtet.

6.2 Unterschiede zwischen JaML und JAST

Die Änderungen am Format bestehen im Einzelnen aus folgenden entfernenden Regeln:

- Entfernen der Textknoten aus dem Dokument, da diese Informationen redundant in den umfassenden Elementen gespeichert sind.
- Entfernen von formatierungsspezifischen Elementen aus JaML. Das sind Elementen für whitespaces, newlines, Kommentaren und Informationen zur Positionierung der Elemente im Quellcode (die Attribute line, col und pos)
- Entfernen der Operatorsymbole, da die Operatoren auch in den entsprechenden unären, binären und ternären Ausdrücken abgelegt sind.
- Entfernen weiterer Symbole, die durch die Java-Sprachdefinition und die Syntax festgeschrieben sind.
- Entfernen aller Schlüsselwörter, da diese Informationen bereits in den entsprechenden Elementen enthalten sind.
- Entfernen der **import**-Anweisungen, da alle Klassen in Java auch direkt durch vollqualifizierte Klassennamen angesprochen werden können.

- Entfernen der Identifier, da diese Informationen ebenfalls redundant in den entsprechenden Elementen gespeichert sind.
- Entfernen der Klammerungen in Ausdrücken. Diese muss dann beim Herausschreiben nach Java wiedereingefügt werden. Dieser Vorgang wird in Abschnitt 6.4.2 beschrieben.

Weiter werden folgende Regeln, die die Struktur des XML Baumes verändern, angewendet:

- Austauschen des Wurzelknotens durch ein <jast:JAST>-Element und die Aktualisierung der Position des Schemas.
- Die Elemente für die einzelnen Modifier wurden zu einem <modifier>-Element zusammengefasst und besitzen die Art des Modifiers nun als Attribut.
- In **if**-Anweisungen werden die Elemente für then- und else-clause entfernt, da bereits über Reihenfolge und Anzahl der Knoten diese Zuordnung möglich ist.
- Vereinfachung des Dokuments durch Entfernen redundanter Elemente, wie den <expression>-Elementen, den <statement>-Elementen und den <type-definition>-Elementen, die jeweils alle entsprechenden Elemente dieser Kategorie umfassen.
- Das <package-declaration>-Element umfasst nun den kompletten Inhalt einer Datei.
- In JaML sind Variablendeklarationen mit einer Liste von deklarierten Variablen zulässig. Diese werden in JAST in einzelne Variablendeklarationen umgewandelt. Ebenso wird mit den Deklarationen von Feldern in Klassen verfahren. Dies ändert nicht die Semantik der entsprechenden Elemente, die Syntax wird nur auf Wesentliches konzentriert.

Diese Liste der Änderungen ist nicht vollständig, aber die wichtigsten Abstraktionen sind dargestellt.

6.3 Transformation von JaML nach JAST

Die Implementierung von JAST baut auf der Implementierung von JaML auf und verwendet als Zwischenschritt das Format von JaML. Die weiteren Bearbeitungsschritte werden dann durch zwei XSL-Transformationen (XSLT) umgesetzt.

Durch das Stylesheet `remover.xsl` werden zunächst alle Modifikationen der ersten Liste aus Abschnitt 6.2 umgesetzt.

Alle weiteren Schritte werden dann durch das XSLT `JAST.xsl` durchgeführt.

Probleme ergaben sich weniger bei der Implementierung dieser Schritte. Erst die Rückrichtung von JAST nach Java wurde aufgrund der entfernten Elemente schwieriger.

Auch die Erstellung von `JAST`-Dateien wird durch das Eclipse Plug-in sowohl auf der Kommandozeile als auch in der IDE unterstützt.

6.4 Transformation von JAST nach Java

Diese Transformation ist im Vergleich zu der Transformation aus Kapitel 5.5 komplizierter. Das liegt an der höheren Abstraktionsstufe der `JAST`-Dateien gegenüber der `JaML`-Dateien. Trotzdem wurde die Transformation ebenfalls komplett in XSLT programmiert.

Zuerst sollte der entstehende `Java`-Code nicht zu unleserlich sein und wurde zumindest durch Einrückung formatiert. Dazu existieren Funktionen in der Datei `functions.xsl`, die diese Einrückung übernehmen. Weitere Funktionen übernehmen das Einfügen von syntaktisch wichtigen Zeichen, wie z.B. Kommata, Strichpunkten und &-Zeichen. Eine weitere Funktion fügt rekursiv die eckigen Klammern für die Deklaration von Arrays ein.

Die Ausdrücke werden durch die Templates aus der Datei `expressions.xsl` bearbeitet. Hier müssen nun wiederum Klammern gemäß der Operatorhierarchie von Java eingefügt werden. Dieser Vorgang wird im Abschnitt 6.4.2 genauer beschrieben, da er auch für die spätere Umwandlung von GAST nach Java notwendig ist.

Die Anweisungen und restlichen Elemente werden durch die Templates der Datei `JAST2Java.xsl` bearbeitet. Bei der Extraktion der Typangaben aus JAST stellte sich das Problem, dass alle Typen qualifiziert werden mussten. Dabei existieren jedoch einige Ausnahmen, die im Abschnitt 6.4.1 beschrieben werden.

6.4.1 Angabe der Qualifizierung der Typen

Einer der Abstraktionsschritte von JaML nach JAST war das Entfernen der **import**-Anweisungen. Dadurch müssen nun bei der Transformation von JAST nach Java sämtliche Typen qualifiziert werden. Ausnahmen existieren wiederum auch hier.

Eine dieser Ausnahmen ist, wenn in einer Typdefinition direkt dieser Typ verwendet wird. Reflexive Assoziationen in UML Klassendiagrammen (siehe [1]) sind ein typisches Beispiel für diese Ausnahme. Der Fall einer reflexiven Assoziation wird im Template abgeprüft und dabei nur der Klassenname als Typname in den entstehenden Code eingefügt.

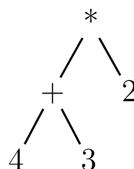


Abbildung 6.1: Baum zum Ausdruck (4+3)*2

6.4.2 Wiedereinfügen der Klammern bei der Transformation nach Java

Im Verlauf der Abstraktion war ein Schritt von JaML nach JAST das Entfernen der Klammern in Form der *parenthesis-expression*-Elemente. Diese Transformation ist ohne Informationsverluste möglich, da diese Informationen durch die Struktur der Instanzdokumente gegeben sind. Beispielsweise wird der Ausdruck `(4+3)*2` zum Baum aus Abbildung 6.1. In diesem Baum ist bereits die notwendige Hierarchie der Operatoren eingebettet und dadurch ist es möglich, die fehlenden und auch notwendigen Klammern wieder einzufügen. Durch die im Folgenden beschriebenen Schritte wird nur die notwendige Klammerung durchgeführt, es wird quasi eine Minimierung der Klammerpaare durchgeführt.

Operatoren unterscheiden sich bekanntlich in zwei Eigenschaften:

1. Die Priorität des Operators ermöglicht die Festlegung der Auswertungsreihenfolge während des Parsens.

2. Die Assoziativität des Operators bestimmt die Reihenfolge der Auswertung von Ausdrücken mit gleichen Operatoren.

Das Wiedereinfügen der korrekten Klammerung bei der Transformation von JAST nach Java entspricht der Umkehrung des Parsevorgangs. Der Hauptteil der Implementierung dieses Codes befindet sich im Verzeichnis `xslt/JAST2java` in der `XSLT`-Datei `bracesForExpressions.xslt`.

Das Template **needsBraces** berechnet hierbei, ob geklammert werden muss. Es besitzt drei Parameter:

- Den Operator des aktuellen Ausdrucks
- Den Operator des enthaltenen Ausdrucks
- Die Position *pos* des enthaltenen Ausdrucks. Mögliche Werte sind “left” und “right”. Bei unären Operatoren ist ebenfalls der Wert “unary” zulässig.

Zunächst werden die Prioritäten der beiden Operatoren durch das Template **priorityOfOperator** und die Assoziativität des aktuellen Operators durch das Template **associativityParent** berechnet. $prio_p$ ist hierbei die Priorität des aktuellen

Operators und $prio_c$ die Priorität des Operators des enthaltenen Ausdrucks, $asso_p$ ist die Assoziativität des aktuellen Operators. Mögliche Werte für die Assoziativität sind “lr” und “rl”.

Klammern werden eingefügt, wenn eine der folgenden Bedingungen erfüllt ist:

- $prio_p > prio_c$
- $prio_p = prio_c$ und $asso_p = lr$ und $pos = right$
- $prio_p = prio_c$ und $asso_p = lr$ und $pos = right$ bzw. $pos = unary$

In allen anderen Fällen werden keine Klammern benötigt und somit nicht eingefügt.

Das Template **extractOperator** wird verwendet um Operatoren aus JAST-Elementen zu extrahieren, die nicht direkt einen Operator repräsentieren (z.B. ein Qualifier) oder die den Operator in einem Attribut beinhalten (z.B. binary expressions). Dazu wird dieses Template rekursiv aufgerufen, bis ein entsprechender Operator extrahiert wurde.

6.5 Test der Transformationen zwischen JAST und Java

Auch für diesen Test bot sich der Inhalt der Datei `src.zip` an.

Zunächst wurden alle enthaltenen Java Dateien in JaML Form gebracht, dann nach JAST transformiert und gegen das Schema `src/schema/JAST/JAST.xsd` validiert. Anschließend wurden die entstandenen `JAST`-Dateien inklusive der entsprechenden Verzeichnishierarchie kopiert und in der Kopie zurück nach Java transformiert. Danach wurden diese `Java`-Dateien mit dem Java Compiler übersetzt um die Compilierbarkeit zu prüfen.

Dieser Test wurde von der vorliegenden Implementierung durchlaufen und bestanden.

Kapitel 7

GAST - Die Implementierung

Dieses Kapitel beschreibt die Königsdisziplin dieser Diplomarbeit, die Erstellung von `GAST`-Instanzen aus `Java`-Projekten und die Umkehrung dieser Transformation. Dabei wird auf den bisher erzeugten Formaten JaML und JAST aufgebaut und diese weiter abstrahiert.

7.1 Erstellung des GASTs

Die Transformation von $JAST \mapsto GAST$ erwies sich vom Aufwand der Implementierung her als die komplexeste. Zur Vereinfachung unterteilt sich die Implementierung in zwei Schritte (siehe Abbildung 7.1), die in diesem Kapitel beschrieben werden.

Als Eingabe erwartet dieser Transformation eine Reihe von JAST-Instanzen, deren Erzeugung beschreiben die Kapitel 6 und 5.

Der erste Schritt wird hierbei pro `java`-Datei durchgeführt, der zweite einmalig pro Projekt.

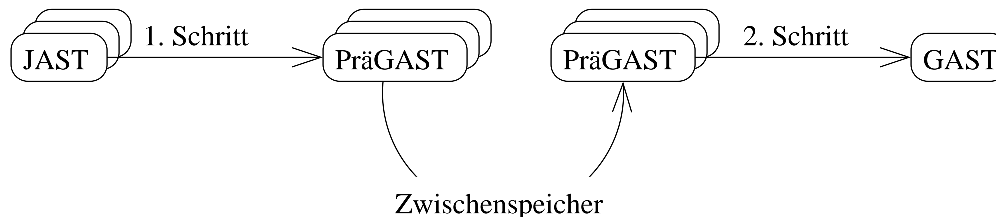


Abbildung 7.1: Schritte zur Erstellung des GASTs aus JAST-Instanzen

7.1.1 Schritt 1: Erstellen der PräGAST-Dokumente

Zunächst werden die Identifier für Typen, Methoden, Felder, etc. aus der behandelten JAST-Instanz von Informationen über ihre Ausprägungen befreit. Da-

zu werden aus den binären Namen durch einen regulären Ausdruck <.*> alle Informationen einschließlich der Zeichen < und > entfernt. Diese Modifikation ermöglicht später eine Zuordnung der verschiedenen Ausprägungen desselben generischen Typs zu diesem Typ. Statt Typen für **Vector<Integer>** und **Vector<String>** wird so nur der generische Typ **Vector<T>** angelegt. Die entsprechend angegeben ausprägenden Typen werden bei den *TypeReference*-Elementen angegeben.

Anschließend dient eine Transformationen zwischen JAST-Instanzen zur Vorbereitung der Haupttransformation. Diese erste XSL Transformation ersetzt akkumulierende Operatoren durch die entsprechende ausführliche Schreibweise. Diese $JAST \rightarrow JAST$ Transformation wird durch die XSL Transformation aus der Datei xslt/ JAST2GAST/CleanAssignmentExpressions.xsl durchgeführt.

Unter akkumulierenden Operatoren versteht man in Java folgende 11 Operatoren: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= und >>>=. Diese können ohne semantische Änderungen immer in eine Zuweisung und den entsprechenden nicht-akkumulierenden Operator umgeschrieben werden. Als Beispiel wird aus a += 2 der semantisch äquivalente Code a = a + 2.

Abschließend wird diese normalisierte JAST-Instanz durch eine weitere XSL Transformation in eine PräGAST-Instanz gewandelt. Für diese Dokumente existiert kein XML Schema, da sie nur als Zwischenergebnisse erzeugt werden. Zu finden ist diese Transformation in der Datei xslt/JAST2GAST/JAST2GAST.xsl.

Diese Transformation entfernt zunächst alle Elemente der JAST-Instanz außerhalb der *package-declaration*-Elemente, da die GAST-Instanz erst im zweiten Schritt aus den Einzelfragmenten zusammengesetzt wird. Die in den *package-declaration*-Elementen enthaltenen Element werden dann in GAST-Elemente umgewandelt, soweit dies möglich ist.

Die folgenden Abschnitte beschreiben Ausnahmen und Besonderheiten bei der Transformation zwischen JAST-Instanzen und GAST-Instanzen.

Umsetzungen in sprachspezifische Artefakte

jast:annotation-definition-, *jast:annotation-modifier-*, *jast:assert-statement-* und *jast:synchronized-statement*-Elemente sowie der Modifier *synchronized* werden allesamt vom GAST nicht unterstützt. Die Transformation verwirft diese Informationen jedoch nicht, sondern wandelt diese Elemente in sprachspezifische Fragmente um. Diese werden in *SAST_Fragment*-Elementen gespeichert.

Generierung der *id*s für Konstanten in Aufzählungen

Bei *enum-constant-declaration*-Elementen müssen die *id*-Attribute neu berechnet werden. Dazu werden die Werte der *id*-Attribute der Aufzählung und der jeweiligen Konstanten konkateniert.

Behandlung von *jast:cast-expression*- und *jast:instanceof-expression*-Elementen

Diese Elemente werden in binäre Ausdrücke mit entsprechendne Operatoren transformiert.

Anpassen der Literale

Konform zur Spezifikation des GAST Modells werden an Literalen für Zeichen und Zeichenketten die begrenzenden Zeichen entfernt.

Übersetzen der Operatoren

Die Operanden werden in ihre GAST Gegenstücke übersetzt. Besondere Aufmerksamkeit benötigen die Prä- und Postfixoperatoren, sie werden gemäß den Vorgaben aus Abschnitt 4.1 ersetzt. Der unäre Operator "+" wird entfernt.

Transformation des Datentyps *Array*

Lesende und schreibende Zugriffe auf Java-Arrays werden auf entsprechende Methoden des GAST-Arrays abgebildet. Im Einzelnen sind dies die Elemente *jast:array-access-expression* und *jast:array-creation-expression*. Der Zugriff auf das Attribute *length* von Arrays wird in einen entsprechenden Methodenaufruf umgesetzt.

Die Initialisierung mehrdimensionaler Arrays mit *jast:array-creation-expression*-Elementen wird umgesetzt zu einem Aufruf einer speziellen generischen Funktion. Der Typparameter dieser Funktion ist der Basistyp des Arrays, der erste Parameter ist die Anzahl der Dimensionen, die folgenden Parameter sind die Größen der jeweiligen Dimension.

Anpassen der Sprungmarken

Die Sprungmarken der beiden Elemente *BreakStatement* und *ContinueStatement* wurden angepasst. Ebenso wurden die *labeled-statement* Elemente zu Labeln an den entsprechenden Anweisungen umgesetzt.

Transformation anonymer Klassen in innere Klassen

Die *jast:anonymous-class-definition*-Elemente, die nur als direkte Kinder von *jast:instance-creation*-Elementen auftreten, wandelt die Transformation um in entsprechende Objetkinstantiierungen einer neu angelegten inneren Klasse. Der Name der Klasse wird aus der ID der anonymen Klasse gebildet und ist somit auch eindeutig. Der Rumpf der anonymen Klasse wird zum Rumpf der neuen inneren Klasse.

Diesen umstrukturierten DOM Baum bezeichnet man als PräGAST-Instanz. Zur Zwischenspeicherung des Baums dienen Klassen, die die Schnittstelle **GAST-StorageInterface** implementieren.

Momentan existiert nur Klasse **GASTMemoryStorage** , die dieses Interface implementiert. Diese Klasse hält die die PräGAST-Instanzen im Speicher vor. Andere denkbare Implementierungen könnten die PräGAST-Instanzen in temporäre Dateien auslagern um Hauptspeicher zu sparen.

7.1.2 Schritt 2: Erstellen des GAST-Dokuments

Dieser Schritt kombiniert die in Schritt 1 zunächst vorbereiteten PräGAST Dokumente zur eigentlichen GAST-Instanz.

Die folgenden Abschnitte beschreiben die notwendigen Zwischenschritte.

Zusammenführen aller PräGAST-Dokumente

Die PräGAST Dokumente besitzen als Wurzelknoten ein *PackageDeclaration*-Element, welches den Namen des Pakets in qualifizierter Form als Attribut enthält. Dieser Paketname wird extrahiert und in die einzelnen Pakete aufgeteilt. Wenn nötig, werden diese Pakete als *PackageDeclaration*-Elemente im Programmteil des Zieldokuments erstellt und der restliche Inhalt des PräGASTs in dieses Paket eingefügt. Auf diese Art und Weise wird die Verzeichnishierarchie des Java-Programms in das GAST-Dokument übernommen.

Einfügen der Definitionen der Basisdatenstrukturen

Als nächstes werden die Deklarationen der externen Basisdatenstrukturen in den Bereich für externe Typdeklarationen (das Element *ExternDeclarationList*) eingefügt. Diese Deklarationen sind im Verzeichnis `xslt/JAST2GAST` in den XML-Dateien `ArrayDeclaration`, `MapDeclaration` und `ListDeclaration` abgelegt.

Behandeln von lokalen Klassen

Unter lokalen Klassen versteht man Klassen, die in einer Methode deklariert wurden und auch nur dort sichtbar sind. Dies bereitete bereits in Abschnitt 5.3.3 Probleme in Verbindung mit ihrem voll qualifizierten Klassennamen. Lokale Typen wurden deshalb nicht in die Spezifikation des GASTs aufgenommen.

Trotz dieser Designentscheidung können lokale Klassen nahezu äquivalent in den GAST umgesetzt werden. Die Umsetzung der lokalen Klassen im GAST erfolgt dabei als innere Klasse. Diese Abbildung wurde gewählt, da sie dem Originalcode näher kommt als die alternative Darstellung als eigenständige Klasse direkt im entsprechenden Paket. Keine Probleme treten hierbei mit der Eindeutigkeit des Namens der neuen inneren Klasse auf, da dieser durch die normali-

sierte ID der Klasse ersetzt wird. Referenziert wird die Klasse weiterhin über die entsprechende ID, die nicht verändert wird.

Es können bei dieser Transformation Probleme entstehen. Diese treten auf, wenn in einer inneren Klasse finale Variablen der umfassenden Methode verwendet werden. Listing 7.1 zeigt ein Beispiel. Diese Problem kann durch eine Modifikation der Konstruktoren angegangen werden.

```
1 void b() {
2   final int lvar=0;
3   class b {
4     int field = lvar;
5   }
6 }
```

Listing 7.1: Beispiel für eine problematische innere Klasse

Auflösen der Typreferenzen

Die im *Program*-Element enthaltenen *TypeReference*-Element müssen alle auf im GAST enthaltene Typen verweisen. In diesem Schritt wird dies sichergestellt und falls notwendig die fehlendenden Typen dem Element *ExternDeclarationList* hinzugefügt.

Die noch nicht aufgelösten Typen sind in den PräGAST-Instanzen mit den Attributen *id*, *package*, *is-interface*, *is-enum*, und *class-name* markiert. Dies sind zugleich die Informationen die benötigt werden, um die Typen zu erzeugen und im richtigen Paket abzulegen.

In einem ersten Teilschritt wird sichergestellt, dass jede Typereferenz nur einmal aufgelöst wird. Anschließend wird für jede dieser restlichen Typreferenzen geprüft, ob diese nicht eventuell bereits im Projekt, aber nicht in der einzelnen Klasse, deklariert wurde. Wenn ja kann sie übergangen werden. Andernfalls wird der Typ dem Element *ExternDeclarationList* in dem richtigen Paket hinzugefügt. Die Markierungsattribute werden abschließend von den *TypeReference*-Elementen entfernt.

Auflösen der Referenzen auf Typparameter

Die im *Program*-Element enthaltenen *TypeParameterBinding*-Elemente müssen analog zu den Typreferenzen auch auf *TypeParameterDeclaration*-Elemente verweisen. Dieser Teilschritt sichert diese Bedingung.

Noch nicht aufgelöste Referenzen sind wieder mit Markierungsattributen versehen, diese sind *pos* und *type*.

Wiederum prüft die Implementierung zuerst, ob die Referenz projektweit aufgelöst werden kann, oder ob sie zu einem Typ gehört, der in die *ExternDeclarationList* eingefügt wurde. Ist dies der Fall, so wird zunächst die Typdeklaration

über das Attribut *type* bestimmt und über das Attribut *pos* die *id* des entsprechenden Typparameters. Diese wird dann im behandelten *TypeParameterDeclaration*-Element als neues Attribut *ref* eingesetzt. Die Markierungsattribute werden abschließend von den *TypeParameterBinding*-Elementen entfernt.

Auflösen der Referenzen auf Felder

Nun wird sichergestellt, dass alle im GAST befindlichen *FieldAccess*-Elemente auf existierenden Felder verweisen. Nicht aufgelöste Referenzen sind nun durch die Attribute *declaring-id* und *name* markiert.

Zunächst wird geprüft, ob das Feld nicht bereits innerhalb des Projekts aufgelöst werden kann. Ist dies nicht der Fall wird der deklarierende Typ in der *ExternDeclarationList* bestimmt und dort ein neues Feld mit Namen, Typ und korrekter ID angelegt. Die Markierungsattribute werden abschließend von den *FieldAccess*-Elementen entfernt.

Auflösen der Referenzen auf Methoden

Ebenso wie mit den Feldern wird mit den Methoden vorgegangen. Die Markierungsattribute sind hier *isStatic* und *name*.

Zuerst werden die *FunctionInvocation*-Element gefiltert um jede fehlende Referenz nur einmal aufzulösen. Danach wird für jede dieser Referenzen geprüft ob sie nicht projektweit auflösbar sind. Wenn nicht identifiziert das Attribut *declaring-id* den Zieltyp eindeutig.

Behandeln von raw types

Bei der Verwendung von so genannten raw types in Java 1.5 kann es Typreferenzen kommen, die nicht auflösbar sind. Diese verweisen dann auf den Typparameter der eigentlich generischen Datenstruktur. Um diese unerfüllten Referencen zu vermeiden, werden diese zu Referencen auf die Klasse **java.lang.Object** umgesetzt.

Einfügen von Labels an Break- und Continue-Anweisungen

Die Spezifikation des GASTs setzt vorraus, dass alle Break- und Continue-Anweisungen ein Label besitzen müssen. Diese Labels dienen zur präzisen Angabe der Ziele dieser Anweisungen.

Um diese Labels einzufügen, wird zunächst ein Markerattribut *undefinedTarget* zur Identifikation herangezogen. Anschließend wird zu jeder gefundenen Anweisung die Zielanweisung bestimmt. Laut [15] sind dies für Continue-Anweisungen die GAST-Elemente *ConditionControlledLoop*, *ForLoop* und *ForeachLoop*. Für Break-Anweisungen kommt zusätzlich das Element *SwitchStatement* hinzu. Eventuell bereits an den Ziel-Anweisungen vorhandene Labels werden weitergenutzt

und falls vorhanden als Ziel an die behandelte Anweisung gehängt. Existiert kein Label, so wird ein neues erzeugt und an beiden Anweisungen verwendet.

Einfügen der Position des Schemas

In den Optionen zum GAST kann angegeben werden, ob und welches Schema eingebunden werden soll.

Dieser Schritt bindet entsprechend der Vorgaben dieses Schema ein.

Bereinigen der id- und ref-Attribute

Abschließend werden die *id*- und *ref*-Attribute in eine Form gebracht, die den Datentypen xs:ID und xs:IDRef entspricht. Dazu werden die Zeichen '`/`', '`~`', '`;`', '`:`', '`(`', '`)`', '`#`', '`.`', '`[`' und '`$`' durch '`_`' ersetzt.

7.2 Transformation von GAST nach Java

Die Transformation von GAST nach Java erstellt aus einer `GAST`-Datei nicht nur eine einzelne `java`-Datei, sondern ein vollständiges Projekt. Daher werden als Eingaben neben der `GAST`-Datei weitere Informationen benötigt:

- Der Namen des Zielprojekts im aktuell Eclipse Workspace
- Die Information, ob als Liste der geworfenen Ausnahmen an Methoden immer "throws java.lang.Throwable" angefügt wird.

Diese Informationen sind der Transformation von externer Seite mitzugeben.

Um die Komplexität der Transformation zu reduzieren wurde sie wiederum in mehrere Schritte unterteilt. Diese Schritte werden nun erläutert.

7.2.1 Anwenden der XSL Transformation

Die verwendete XSL Transformation befindet sich in der Datei `GAST2Java.xsl` im Verzeichnis `xslt/GAST2Java/`.

Hauptaufgabe ist die Umsetzung der GAST-Elemente in semantisch äquivalenten Quellcode der Sprache Java. Eine weitere wichtige Aufgabe der Transformation ist das Einfügen von Hinweisen zu Programmstart und -ende, Anfang und Ende des Inhalts von Paketen und Klassen. Diese Informationen werden im nächsten Schritt genutzt.

Die importierte Datei `ArrayHandling.xsl` übernimmt die Umsetzung des GAST-Arrays nach Java, indem alle Methoden der Klasse Array des GASTs in äquivalenten Java-Code umgesetzt werden.

In der Datei `bracesForExpressions.xsl` befindet sich der XSLT Code, der analog zu Abschnitt 6.4.2 die benötigten Klammern wieder in die Ausdrücke einfügt.

Allgemeine Funktionen befinden sich in der Datei `functions.xsl`. Diese dienen zum Einfügen syntaktisch zwingend vorhandener Elemente, wie Kommata in Listen oder den &-Zeichen zum Trennen mehrerer Einschränkungen bei generischen Typen.

Auf die Transformationen zur Umsetzung wird hier nicht im Detail eingegangen, da diese meistens sehr direkt gelöst werden konnten. Die Ausnahmen beziehungsweise besonders interessante Fälle werden jetzt kurz angerissen.

Eine Besonderheit der Umsetzung betrifft die Literale *BaseTypeLiteral* für Zeichenketten (Typ `String`) und einzelne Zeichen (Typ `char`). Dort müssen die begrenzenden Zeichen (“ ” sowie ‘ ’) wieder eingefügt werden.

Ein interessanter Teil der Transformation ist das Template zur Ersetzung von *gast:TypeReference*-Elementen. Diese Elemente enthalten alle Informationen, die zur Darstellung des korrekten Java-Typen notwendig sind. Da die Darstellung von GAST-Arrays ebenfalls über *TypeReference*-Elemente erfolgt (siehe Abbildung 7.2), müssen die Dimensionen des Arrays durch rekursive Anwendung des Templates bestimmt werden und an den Basistyps des Arrays angehängt werden. Sind alle Typreferenzen auf Arrays abgearbeitet, so wird der Basistyp bearbei-

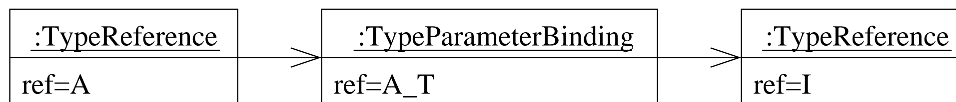


Abbildung 7.2: Objektdiagramm zur Umsetzung von `int[]` im GAST

tet. Der Name des Typs wird hierbei über die Referenz auf seine Deklaration bestimmt. Anschließend werden die ausprägenden Typen über die *TypeParameterBinding*-Elemente bestimmt und in den Java-Quellcode eingefügt.

In zwei weiteren Templates werden unäre und binäre Operatoren zwischen GAST und Java übersetzt, da sich diese zwar stark ähneln aber teilweise doch verschieden sind. Unbekannte Operatoren werden durch eingebettete SAST Fragmente beschrieben (siehe Abschnitt 3.14). Diese werde hier, falls es sich um Java Fragmente handelt, bereits direkt wieder in Java übersetzt. Operatoren aus anderen Sprachen werden in einen Blockkommentar eingeschlossen. In diesem Kommentar befindet sich die Zeichen für den Operator und die Quellsprache.

Diese beiden Templates leisten neben dem richtigen Umsetzen der Operatoren zwischen GAST und Java auch die notwendige Klammerung der Unterausdrücke. Zusätzlich ist das Template für *BinaryExpression* verantwortlich, die in Abschnitt 4.1 ersetzten Inkrement- und Dekrementoperatoren zurück zu übersetzen.

Diese Umsetzung der Operatoren erfordert das Erkennen größerer Muster im XML Baum. Diese Muster sieht man in den Abbildungen 7.3 und 7.4.

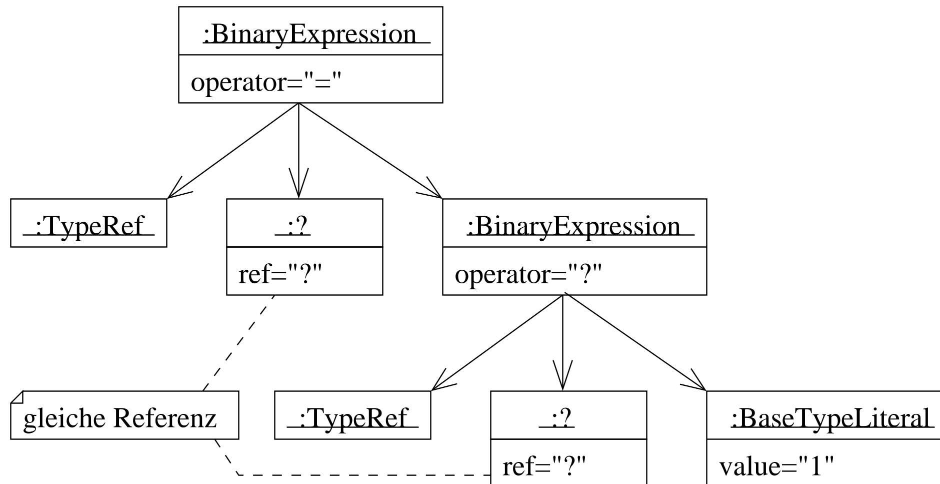


Abbildung 7.3: Objektdiagramm für einen Präfixoperator im GAST

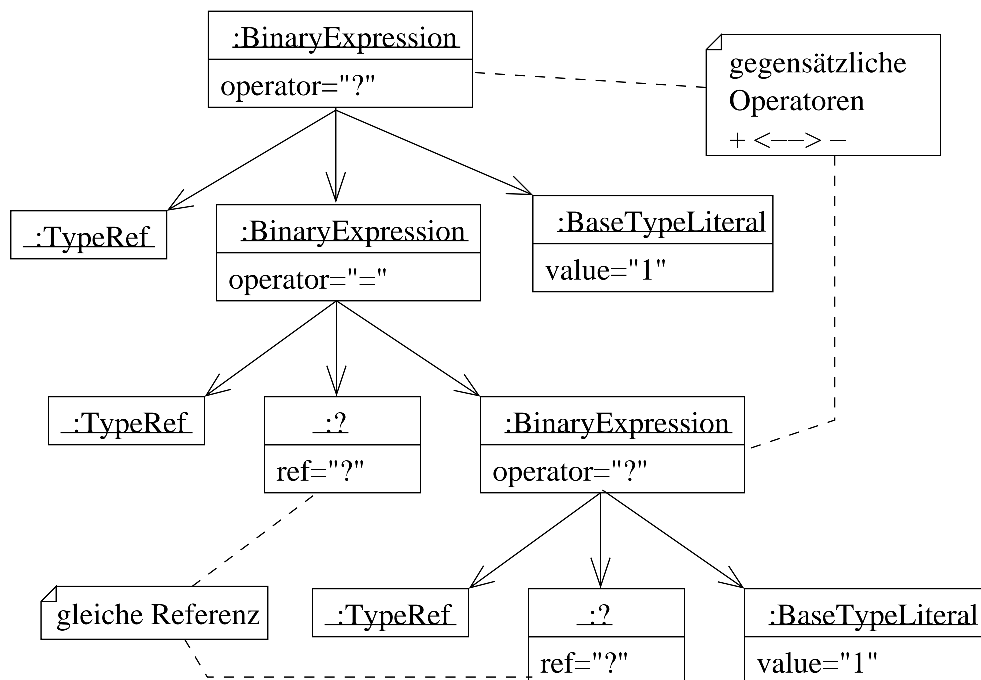


Abbildung 7.4: Objektdiagramm für einen Postfixoperator im GAST

Die Methoden `testForPostfixExpression` und `testForPrefixExpression` dienen zum Erkennen dieser speziellen Muster der umgesetzten Operatoren. Abhängig vom Ergebnis dieser Test wird der entsprechende Operator (++ bzw. −−) bestimmt und anschließend geprüft, ob Klammerung notwendig ist.

Da im GAST nicht zwischen dem Aufruf von Konstruktoren und Operationen unterschieden wird, Java aber Konstruktoren mit dem `new`-Operator aufruft, muss das entsprechende Template diese Unterscheidung durchführen. Aufgrund der Definition des GASTs ist im Attribut *isConstructor* die notwendige Information enthalten. Über die Referenz, die als Attribut jedem *FunctionInvocation*-Element zugeordnet ist, lässt sich leicht auch auf die Deklaration der Operation zugreifen.

SAST_Fragment-Elemente repräsentieren Quellcode (siehe Abschnitte 1.6 und 3.14), der nicht in den GAST umgesetzt werden kann. Die Betrachtung der sprachspezifischen Fragmente aus den GAST-Instanzen erfolgt abhängig der Quellsprache. JAST-Fragmente werden hierbei durch die in Abschnitt 6.4 beschriebene Transformation behandelt. Fragmente anderer Sprachen werden als Blockkommentar umgesetzt.

Für den nächsten Schritt fügt die Transformation auch Informationen über die Struktur des Projekt in Verzeichnissen und Dateien ein. Diese Informationen befinden sich in Zeilenkommentaren folgender Form:

- `//__START__PROGRAM__`: Anfang des Programms
- `//__END__PROGRAM__`: Ende des Programms
- `//__START__PACKAGE__ name=packageName`: Anfang des Inhalts eines Pakets
- `//__END__PACKAGE__ name=packageName`: Ende des Inhalts eines Pakets
- `//__START__FILE__ name=fileName`: Anfang des Inhalts einer Datei
- `//__END__FILE__ name=fileName`: Ende des Inhalts einer Datei

Eine direkte Umsetzung dieser Funktionalität ist aufgrund der Beschränkungen von XSLT 1.0 derzeit nicht möglich. Mit XSLT 2.0 oder durch Erweiterungen des XSL-Transformers wäre es eventuell jedoch möglich ohne weiteren Java Code auszukommen. Dieser Ansatz wurde hier nicht weiter verfolgt.

Das Ergebnis dieses Schrittes ist eine Datei, welche allen Java-Code des Zielprojekts enthält. Eine Ausnahme stellen die Paketdeklarationen dar, die in Java `package pkgName;` geschrieben werden.

7.2.2 Erstellen des Java Projektes

Die Umsetzung der im ersten Schritt generierten Datei in ein Java-Projekt mit korrekter Anordnung der Java-Dateien in Verzeichnissen wird nun durchgeführt.

Dazu nutzt die Klasse **GASTFileParser** die von Eclipse bereitgestellten Klassen zur Verwaltung von Projekten, Verzeichnissen und Dateien. Dies ist jedoch nicht zwingend notwendig, eine Implementierung nur mit Java-Klassen des Standardumfangs ist ebenfalls möglich. Ein Parser analysiert die Zeilenkommentare aus Schritt 1 und legt entsprechend Verzeichnisse und Dateien an. Die Dateien erzeugt er in den entsprechenden Verzeichnissen, die die Pakete repräsentieren.

Im Unterschied zu dem Originalquellcode, bei dem mehrere Klassen neben maximal einer öffentlich sichtbaren Klasse in einer Datei auftreten konnten, entsteht nun für jede Klasse eine eigene Datei mit dem Namen der Klasse. Dies verändert die Semantik des Programms nicht, ebenfalls treten keine Probleme mit der Sichtbarkeit dieser Klassen auf.

Das Ergebnis dieser Transformation ist jedoch nicht immer ein kompilierbares Java-Projekt, da keine Informationen über den Classpath des Projekts gespeichert wurden. Diese externen Bibliotheken müssen wieder im Projekt angegeben werden.

Kapitel 8

Handbuch für das Eclipse Plug-in

8.1 Features des Plug-ins

Das im Rahmen der Diplomarbeit entwickelte Plug-in leistet folgende Dienste, sowohl beim Einsatz in Eclipse als auch auf der Kommandozeile:

- Erzeugen von `JaML`-Dateien
- Erzeugen von `JAST`-Dateien
- Erzeugen von `GAST`-Dateien
- Transformation einer `GAST`-Datei zurück in ein `Java`-Projekt.

8.2 Anforderungen zur Verwendung des Plug-ins

8.2.1 Softwareanforderungen

Um das Pflug-in verwenden zu können, sind folgende Minimalanforderungen an die Software zu stellen:

- Java Runtime Environment (JRE) 5.0 oder höher (Downloadbar auf [4])
- Eclipse SDK 3.2 oder höher (Downloadbar auf [2]). Dies ist aufgrund der in älteren Versionen enthaltenen Fehlern [19][20][21][18] notwendig. Diese Fehler wurden in Abschnitt 5.3.3 weiter beschrieben.

8.2.2 Hardwareanforderungen

An Hardware sei auf die Minimalanforderungen von Eclipse verwiesen, die natürlich auch für den Einsatz eines Plug-ins gelten.

Zusätzlich werden für die Hardware folgende Empfehlungen ausgesprochen:

- Prozessor mit mindestens 2 GHz
- Mindestens 1 GB Hauptspeicher

8.3 Installation des Plug-ins

Wenn Sie das Plug-in bereits in der Form einer `zip`-Datei vorliegen haben, können Sie den nächsten Abschnitt überspringen.

8.3.1 Export des Plug-ins

Diese Schritt-für-Schritt Anleitung setzt voraus, dass das Projekt bereits in Quellform in den aktuellen Workspace von Eclipse eingebunden wurde.

Öffnen Sie zunächst die Datei `plugin.xml` mit dem Plug-in Manifest Editor.

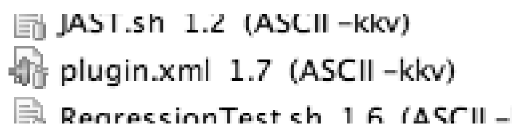


Abbildung 8.1: Öffnen der Datei `plugin.xml`

Wechseln Sie dort in den unteren Tabs auf die Seite "Overview" und starten Sie den Export Wizard.

Abbildung 8.2: Starten des Export Wizards

Wählen Sie dort im Bereich "Available Plug-ins and Fragments" das JaML Plug-in aus. Im Reiter "Destination" wählen Sie die Option "Archive file" und tragen Sie eine geeignete Position für die `zip`-Datei ein. Nach einem Klick auf "Finish" wird das Plug-in in geeigneter Form verpackt. Weitere Schritte sind zum Exportieren nicht notwendig.

8.3.2 Einbinden des Plug-ins in Eclipse

Um das JaML Plug-in in Eclipse einzubinden, genügt es, das Plug-in in Form einer `zip`-Datei direkt in das Wurzelverzeichnis der Eclipse Installation zu entpacken. Sollten dabei Probleme auftreten, kann es helfen, Eclipse auf der Kommandozeile mit der Option `--clean` zu starten.

8.4 Einsatz des Plug-ins

Das entstandene Plug-in beinhaltet sowohl graphische Komponenten in der Eclipse-GUI als auch einen Zugang über die Kommandozeile.

8.4.1 In Eclipse

Das JaML Plug-in steht prinzipbedingt nur für `Java`-Projekte zur Verfügung.

Generierung von `JaML`-, `JAST`- oder `GAST`-Dateien

Soll ein `Java`-Projekt durch das JaML Plug-in umgesetzt werden, so müssen für dieses Projekt die Eigenschaften bearbeitet und in den Optionen des AST Generators die entsprechenden Zielformate selektiert werden (siehe Abbildung 8.3). Die vorgenommenen Einstellungen werden persistent zum Projekt im Workspace gespeichert und müssen nicht mehr vorgenommen werden.

Wurden alle Einstellungen vorgenommen, wird durch einen Klick auf den in Abbildung 8.4 links abgebildeten Button ein Dialog zur Auswahl des Zielprojekts gestartet. In diesem Dialog werden nur bereits geöffnete `Java`-Projekte angezeigt. Das Zielprojekt wird mit dem Dialog aus Abbildung 8.5 ausgewählt. Durch Bestätigen mit einem Klick auf "Ok" wird das Erstellen der gewählten Dateien gestartet.

Die Ergebnisse liegen danach direkt im `Java`-Projekt und werden durch einen automatischen Refresh des Workspaces eingebunden.

Umwandlung von `GAST`-Dateien in `Java`-Projekte

Auch die Umwandlung von `GAST`-Dateien in `Java`-Projekte ist direkt in Eclipse möglich.

Dazu klickt man auf den rechten Button aus Abbildung 8.4. In dem sich öffnenden Dialog (siehe Abbildung 8.6) wählt man den Namen des zu erstellenden Projektes und die `GAST`-Datei aus. Existiert das Projekt bereits, so wird der Name rot unterlegt. Wird der Name nicht geändert, so wird das existierende Projekt gelöscht und mit dem Inhalt der `GAST`-Datei überschrieben.

Nach Abschluss der Operation befindet sich das neue Projekt geöffnet im Workspace und kann direkt weiter bearbeitet werden.

8.4.2 Auf der Kommandozeile

Für die Verwendung auf der Kommandozeile sind vier Skripte im Verzeichnis des `JaML`-Plug-ins enthalten. Diese befinden sich im Verzeichnis `plugins/JaML_*`. Die Wildcard `*` steht hierbei für die verwendete Version des Plug-ins.

Vor dem Einsatz der Skripte muss über die Umgebungsvariable `WORKSPACE` der Workspace für das temporär angelegte Projekt angegeben werden.

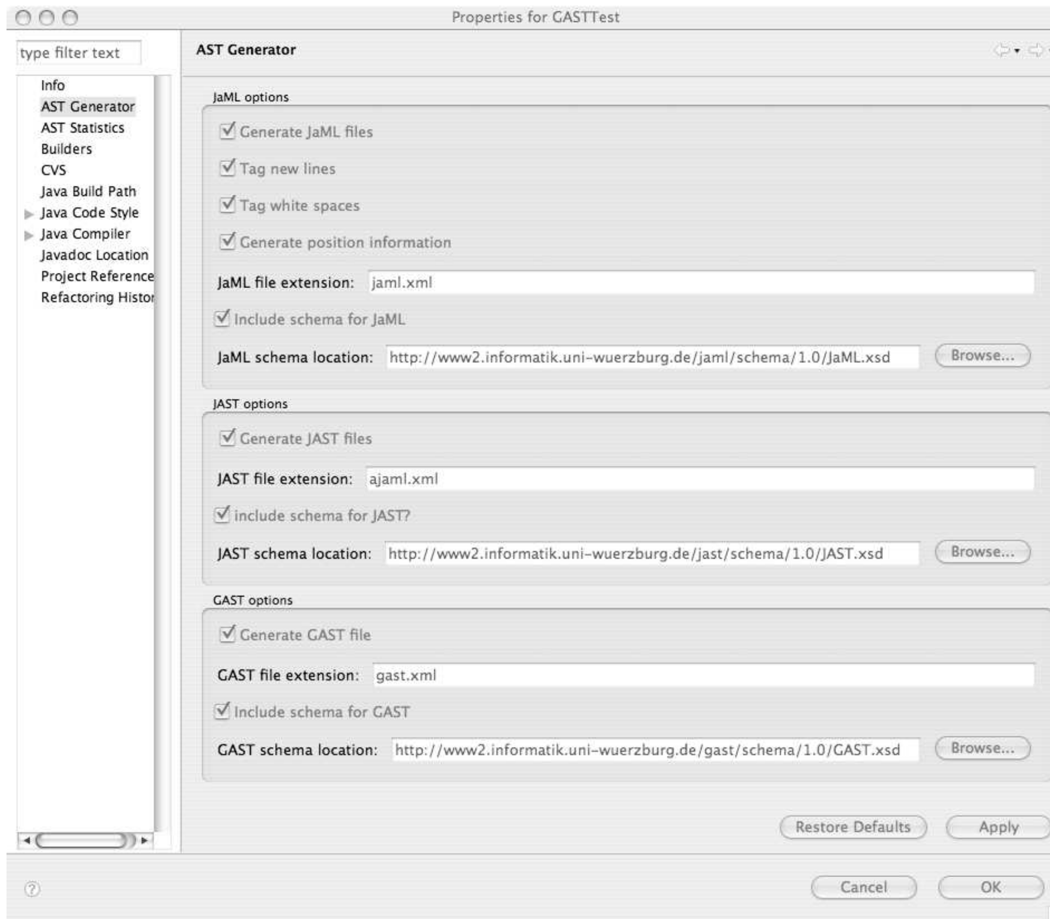


Abbildung 8.3: Optionen des JaML Plug-ins

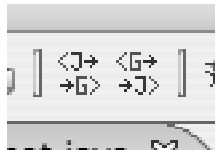


Abbildung 8.4: Icon zum Starten des Plug-ins

Abbildung 8.5: Dialog zur Auswahl des Zielprojekts

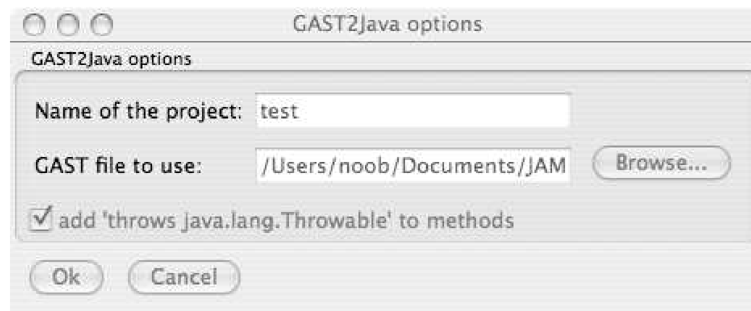


Abbildung 8.6: Optionsdialog für GAST2Java

Drei der Skripte (`JaML.sh`,`JAST.sh` und `GAST.sh`) dienen zur Generierung von Dateien im jeweiligen Format. Diese Skripte werden über Parameter gesteuert, die in Tabelle 8.2 aufgezählt sind. Dort **fett** gedruckte Parameter sind für alle drei Skripte anwendbar, die anderen nur für das Skript `JaML.sh`. Ein Aufruf eines Skripts mit der Option `--help` zeigt jeweilige Abhängigkeiten der Parameter.

Das vierte Skript (`GAST2Java.sh`) dient zur Umwandlung einer `GAST`-Datei in ein Java-Projekt. Die dazu benötigten Parameter findet man in Tabelle 8.1.

Option		Parameter	Beschreibung
kurz	lang		
-h	**−−help**		Gibt die Hilfe zur Kommandozeilenversion aus
-f	**−−file**	<GAST-Datei>	die umzusetzende `GAST`-Datei
-p	**−−project**	<Projektname>	der Name des Zielprojekts
-t	**−−throwsClause**		füge Throws-Klausel ein

Tabelle 8.1: Beschreibung der Kommandozeilenparameter von `GAST2Java.sh`

Option		Parameter	Beschreibung
kurz	lang		
-h	**--help**		Gibt die Hilfe zur Kommandozeilenversion aus
-b	**--baseDir**	<directory>	Gibt das Basisverzeichnis an (das Verzeichnis des *default* Package)
-cp	**--classpath**	<jarfiles>	Setzt den Classpath für JaML, analog zum Setzen des Classpaths in Eclipse
-e	**--extension**	<extension>	Setzt die Dateiendung der JaML Dateien
-x	**--xmlschema**	<file>	Angabe der Schemadefinition, die in die generierten Dateien als Verweis eingefügt wird.
-X	**--noxmlschema**		Unterbindet das Einfügen des Verweises auf die Schema-definition
-p	--positions		Fügt Informationen über die Position der Elemente im Quellcode ein
-w	--whitespace		Markiert Whitespaces durch XML-Tags
-n	--newline		Markiert Zeilenumbrüche durch XML-Tags
-f	**--files**	<files>	Die angegebenen Java Dateien werden als Eingabe verwendet. Die Angabe erfolgt als Pfadangabe relativ zum Basisverzeichnis
-r	**--recursive**		Alle Java Dateien unterhalb des Basisverzeichnisses werden als Eingabe verwendet

Tabelle 8.2: Beschreibung der Kommandozeilenparameter der Skripte zum Erstellen der `XML` Dateien

Kapitel 9

Ergebnis und Diskussion

Diese Diplomarbeit besteht aus 4 großen Teilen und diese sollen einzeln betrachtet werden.

9.1 Implementierung von JaML

JaML war als Weiterentwicklung auf den Stand von Java Version 5 zu bringen. Das bedeutetet die Integration der neuen Sprachelemente und den Test der Umsetzung. Dieser Test wurde in Abschnitt 5.6 beschrieben und von der vorliegenden Implementierung problemlos durchlaufen.

Die im AST von Eclipse aufgetretenen Fehler verkomplizierten diesen Teil leider unnötig, konnten aber alle durch die Entwickler des JDT Projekts kurz nach der Meldung im Bugzilla behoben werden. Auch die Erstellung eines allgemeingültigen Schemas zu den entstehenden `JaML`-Dateien war wegen der hohen Variabilität der Sprache Java nicht ganz trivial.

9.2 Implementierung von JAST

JAST wurde im Rahmen dieser Diplomarbeit entwickelt. Die Abstraktion der Syntax im Vergleich zu JaML, die jedoch die Mächtigkeit der Darstellung nicht beeinflusst, sorgt für eine deutliche Vereinfachung der Instanzdokumente. Dadurch sinkt die Größe der erstellten `XML`-Dateien und auch die Zahl der notwendigen Elemente sinkt. Auf diese Ersparnis wird noch im Abschnitt 9.6 eingegangen, der alle drei Formate und auch das Original vergleichen wird. Jedoch gehen durch diese Abstraktionen keine relevanten Informationen verloren, was durch den in Abschnitt 6.5 beschrieben und durchgeführten Text gezeigt wird.

Probleme traten bei den Transformationen von JaML nach JAST und JAST nach Java in Verbindung mit der Klammerung von Ausdrücken und bei der Qualifizierung von Typen auf. Die Erfahrungen die bei der Lösung dieser Probleme entstanden, wurden bei der Transformationen des GASTs wieder eingesetzt.

9.3 Die Definition des GASTs

Nun kommen wir zu einer der Hauptaufgaben dieser Diplomarbeit, der Definition des sprachunabhängigen Modells für die abstrakten Syntaxbäume. Dieser theoretische Abschnitt nimmt in dieser Ausarbeitung einen hohen Stellenwert, da er eine Grundlage für die Entwicklung weiterer Software sein soll.

Als Vorbereitung auf die Definition des GAST-Modells dienten neben den Erfahrungen mit Java auch die Vorträge des Seminars zum Thema Programmiersprachen im Sommersemester 2006 und das "Taschenbuch Programmiersprachen". Dort wurden neben weiteren objektorientierten Sprachen auch andere Sprachkonzepte präsentiert. Die dann wieder in die GAST Definition einfließen konnten.

Leider war der Prozess der Definition aufwändiger als zunächst gedacht, da nicht nur entschieden werden musste, welche Elemente in die Definition des GASTs aufgenommen werden sollten, sondern auch die Semantik der Elemente war zu definieren.

Durch die Umwandlung in das GAST Format sank sowohl die Größe der `XML`-Dateien als auch die Anzahl der Elemente in diesen Dateien. Genauere Zahlen zu diesem Thema sind im Abschnitt 9.6 zu finden.

Der GAST bietet über die definierten Elemente und Standardtypen hinaus weitere Möglichkeiten zur Repräsentation des Programmcodes. Dazu existiert die Liste der externen Typdeklarationen, die genutzt wird, um primitive Typen und Typen aus Bibliotheken darzustellen. Darüber hinaus können nicht umsetzbare Sprachelemente durch sprachspezifische Fragmente eingebettet werden, sodass dadurch kein Informationsverlust stattfindet. Durch diese Möglichkeiten wird der GAST ausreichend mächtig und flexibel um weitere Sprachen umsetzen zu können.

9.4 Die Implementierung *Java* ↔ *GAST*

Die Implementierung der Transformationen zwischen GAST und Java demonstriert die Verwendbarkeit des GASTs und die Umsetzbarkeit real existierender Programmiersprachen in den GAST und umgekehrt.

Aufgrund desAufwandss bei der Definition des GAST Modells, konnte die Implementierung der Transformationen erst spät begonnen werden und nicht vollständig umgesetzt werden. Hier folgt nun eine Auflistung der nicht umgesetzten Elemente des GASTs und Sprachelemente von Java.

9.4.1 Nicht in Java umgesetzte GAST-Elemente

Zuerst kommt eine Auflistung der GAST Elemente, die nicht umgesetzt werden konnten. Es wurde nur die Basisdatenstruktur Array aus dem GAST heraus

nach Java umgesetzt und nicht auch die GAST-List und die GAST-Map (siehe Abschnitt 3.8). Diese können auf geeignete Implementierungen der Java Schnittstellen **java.util.List** und **java.util.Map** abgebildet werden, die Klassen **Vector** und **HashMap** bieten sich an.

Deklarationen von Funktionen nach Abschnitt 3.9.3 können in Java nicht direkt verwendet werden, sondern müssen als Operationen von Klassen abgebildet werden. Ebenso müssen Variablen von Funktionstypen und Funktionsdeklarationen nach Abschnitt 3.10.6 speziell behandelt werden. Diese erfordern eine Umsetzung als Funktor, siehe dazu [12] Kapitel 5.13.

Im GAST Modell sind bei der Anweisung zur Fallauswahl Bereiche als Ausdruck bei einem Fall erlaubt. DieseMöglichkeitt existiert in Java nicht, kann aber durch vollständige Aufzählung des entsprechenden Bereichs in Verbindung mit entsprechenden Case-Klauseln in Java umgesetzt werden. Der Bereichsoperator kann allgemein leicht in ein entsprechendes Literal eines Arrays umgesetzt werden.

Die im GAST Modell zugelassene Mehrfachvererbung kann nicht direkt in Java umgesetzt werden da Java diese Form der Vererbung nicht unterstützt. Aus diesem Grund muss auf alternative Formen der Umsetzung ausgewichen werden. Eine Möglichkeit könnte wie folgt aussehen. Erbt eine Klasse von n Superklassen, so werden $n-1$ dieser Vererbungen durch eine Delegation an entsprechende Objekte umgesetzt. Um die Zuweisbarkeit zu erhalten, wird für jede dieser Klassen eine Schnittstelle definiert und in die Liste der implementierten Schnittstellen eingetragen. Diesermöglichth nicht eine 100prozentige Umsetzung der Funktionalität, kann aber als erster Ansatz genutzt werden.

Alle weiteren Elemente des GASTs werden in Java umgesetzt.

9.4.2 Nicht in GAST umgesetzte Java Konstrukte

Nach Durchsicht der Syntaxdiagramme aus [12] fielen als Ausnahmen nur die folgenden Java Elemente auf, die durch die hier entwickelten Transformationen nicht in den GAST umgesetzt wurden. Dies lag weniger an Mängeln des GAST Modells sondern mehr an der knappen Zeit gegen Ende der Diplomarbeit. Die Elemente werden nach der Dringlichkeit der Umsetzung sortiert dargestellt.

Die Initializer-Elemente in Form der *instance-* und *static-initializer* werden im GAST nicht direkt unterstützt. Auch eine Umsetzung ihrer Semantik in den GAST ist nicht exakt möglich. Eine näherungsweise Umsetzung der *instance-initializer* besteht im Einkopieren des Code in die einzelnen Konstruktoren des Typs. Statische Initializer werden beim Initialisieren der Klasse ausgeführt (siehe [15] 8.7). Auch diese können näherungsweise in den GAST umgesetzt werden indem auch dieser Code in die Konstruktoren kopiert und sichergestellt wird, das der Code nur einmal ausgeführt wird.

Dies sind die bis zum Ende der Diplomarbeit bekannten Umsetzungslücken. Allen gemeinsam ist, dass der GAST prinzipiell eine Umsetzung zulässt. Ob diese

dann allgemein gültig ist oder nur mit Einschränkungen verwendet werden kann, hängt vom Einzelfall des Codes ab.

Auf jeden Fall werden Typen durch die Darstellung in der Liste der externen Deklarationen korrekt in den GAST abgebildet. Modifier, die nicht bereits im GAST definiert sind, werden durch sprachspezifische Fragmente abgebildet.

9.5 Übersicht über Transformationen

Die Abbildung 9.1 zeigt eine Übersicht über die in den einzelnen Abschnitten der Arbeit verwendeten Transformationstechniken.

Als zentrales Element muss hierbei der AST genannt werden, die Entwicklung eines AST-Generators für Java nicht einfach ist.

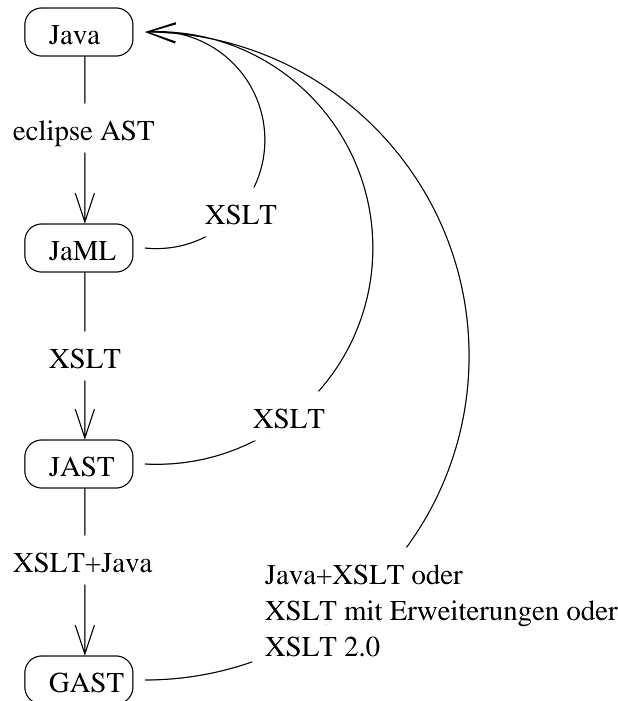


Abbildung 9.1: Chart zum Projekt HelloWorld

9.6 Vergleich der Formate

In diesem Abschnitt werden die drei Formate, die in dieser Diplomarbeit entstanden sind, anhand einiger einfachen Metriken verglichen. Die Metriken sind die Größe der erstellten Datei und die Anzahl der zur Darstellung benötigten Elemente in den XML Dokumenten. Jeweils zum Vergleich wird die Dateigröße der

	filesize (byte)	lines of code	nodes in XML
Java	273	12	-
JaML	85 692	-	600
JAST	62 856	-	304
GAST	13 705	-	340
GAST (zip)	216	-	-

Tabelle 9.1: Werte zum Projekt HelloWorld

original Java-Datei ebenfalls dargestellt. Die Basis der Daten sind zwei Projekte, die auch auf der beigelegten CD-Rom zu finden sind.

9.6.1 Betrachtung des HelloWorld-Projekts

Das obligatorische "Hello World!" gibt dieses Programm zwar nicht aus, aber es begrüßt den Nutzer und gibt die restlichen Kommandozeilenparameter aus.

Man stellt anhand Tabelle 9.1 und Diagramm 9.2 schnell fest, welche Dateigrößen entstehen wenn selbst kleinste Java-Programme umgesetzt werden sollen. Die Größen der JaML-Dateien ergeben sich aufgrund der Markierung aller sprachlicher Elemente (inklusive aller Zeichen, Whitespaces und Newlines). Allerdings ist durch die Transformationen von JaML nach JAST eine Reduktion derDateigrößenn möglich. Dies ergibt sich aus der in Abschnitt 6.3 beschriebenen Transformation, die die Struktur des JaML-Dokuments vereinfacht. Eine stärkere Reduktion ergibt sich durch die Transformation von JAST nach GAST.

Ebenfalls interessant ist die Anzahl der Knoten in den Dateien der einzelnen Formaten. Eine deutliche Vereinfachung zwischen JaML und JAST, bei diesem Beispiel um 50%, wird durch das Entfernen redundanter und überflüssiger Knoten erreicht. Das HelloWorld-Projekt fällt bei der Betrachtung der Knotenzahl und deren Veränderung beim Übergang zwischen JAST und GAST etwas aus der Reihe. Man erwartet eine weitere Reduzierung der Knoten beim Übergang von JAST in den GAST. Diese bleibt aus, da der Aufwand für die Liste der externen Typdeklarationen die Einsparungen wieder auffrisst. Dies ändert sich jedoch im zweiten betrachteten Projekt.

9.6.2 Betrachtung des CodeCollection-Projekts

Diese Projekt ist eine lose Sammlung von Java-Klassen, die in den GAST transformierbar sind und wieder in funktionierenden Java-Code zurück übersetzt werden können.

Wie auch beim ersten Beispiel sieht man hier in Tabelle 9.2 und Diagramm 9.3 die enorme Größe der JaML-Datei und die starke Verkleinerung der JAST- und GAST-Dateien. Die Einsparungen liegen bei ca. 30% von JaML nach JAST

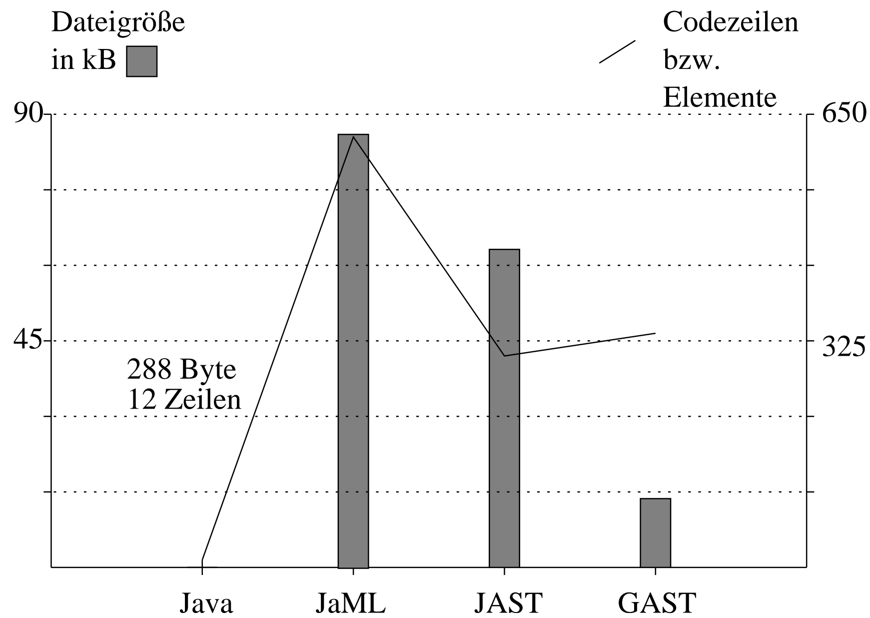


Abbildung 9.2: Chart zum Projekt HelloWorld

	filesize (byte)	lines of code	nodes in XML
Java	10 285	506	-
JaML	2 053 502	-	18 130
JAST	1 384 575	-	7 252
GAST	159 076	-	3 061
GAST (zip)	9 920	-	-

Tabelle 9.2: Werte zum Projekt CodeCollection

und bei über 90% von JaML nach GAST. Ebenfalls sieht man die deutlichen Einsparungen bei der Knotenanzahl, die beim Projekt HelloWorld noch nicht sichtbar wurden. Die belaufen sich auf über 50% zwischen JaML und JAST und auf ca. 80% zwischen JaML und GAST. Diese Einsparungen ergeben sich aus der effizienteren Behandlung der Typreferenzen im GAST. In JaML und JAST werden pro Typreferenz alle Informationen über diesen Typen wiederholt, im GAST lediglich ein einfacher Verweis auf die jeweilige Deklaration des Typs. Diese Deklaration enthält dann alle notwendigen Informationen. Dadurch wird auch sichergestellt,dasss keine Informationen verloren gehen.

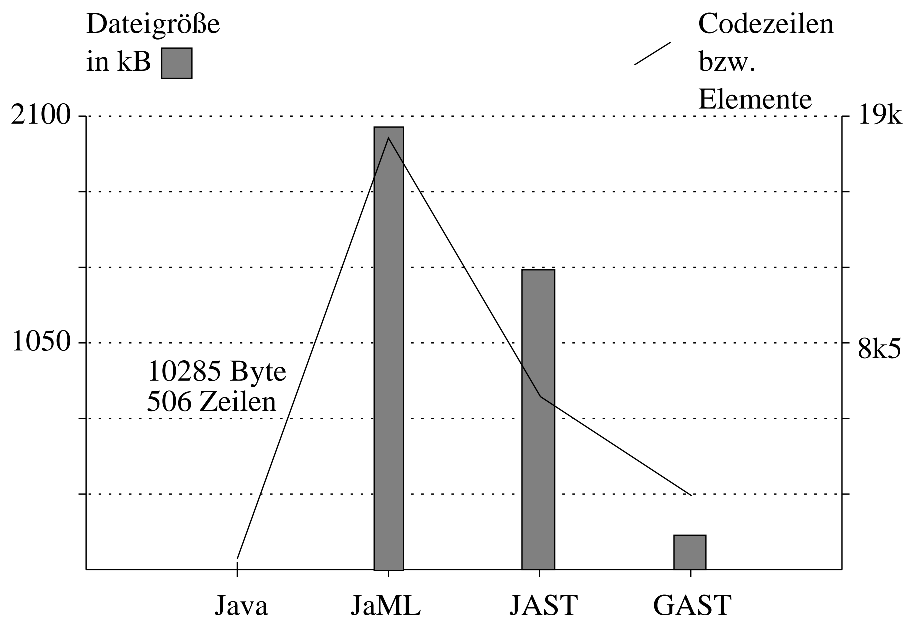


Abbildung 9.3: Chart zum Projekt CodeCollection

9.7 Größenreduzierung der GAST-Dateien

Eine gängige Technik zur Reduzierung des Platzbedarfs bei XML-Datei ist die Kompression in einer zip-Datei. Dieses Verfahren wird unter anderem bei OpenOffice, den neueren Microsoft Office Versionen und zur Kompression von svg-Dateien eingesetzt.

Im Fall der GAST-Dateien liegt die Dateigröße der komprimierten Dateien im Bereich der Dateigrößen der original java-Dateien. Berücksichtig man den deutlich höheren Informationsgehalt des GASTs, so sind kleine Differenzen leicht verschmerzbar.

Daten zu den beiden Beispielprojekten sind in den Tabellen 9.1 und 9.2 aufgezeigt.

9.8 Ausblick

Für die Zukunft ist eine weitere Verbesserung der Java Unterstützung geplant. Diese soll die noch vorhandenen Fehler und auch die fehlenden Umsetzung zwischen GAST und Java ausbügeln.

Auf der Definition des GAST-Modells können weitere Arbeiten anknüpfen. Beispielsweise erarbeitet Per Pascal Grube eine Transformation des GASTs um UML Klassen-, Sequenz- und Aktivitätsdiagramme darzustellen.

Listings

1.1 eine einfache Anweisung 3
3.1 mehrfache Deklarationen in Java 16
3.2 Arrayzugriffe in Lua 16
3.3 Beispiel für einen Block 29
3.4 Beispiel für Variabledeklarationen 31
3.5 Beispiel einer Funktionsdeklarationanweisung 32
3.6 Beispiel für Ausdrucksanweisungen 32
3.7 Beispiel für eine Zweifachauswahl 33
3.8 Beispiel für eine Einfachauswahl 33
3.9 Beispiel für eine Fallauswahl 34
3.10 Beispiel für eine zählende Schleife 36
3.11 Beispiel für eine iterierende Schleife 37
3.12 Beispiel für eine fußgesteuerte Schleife 38
3.13 Beispiel für eine kopfgesteuerte Schleife 38
3.14 Beispiel für ein Throw-Statement 39
3.15 Beispiel für ein Throw-Statement 40
3.16 Beispiel für ein Break-Statement 41
3.17 Beispiel für ein Continue-Statement 42
3.18 Beispiel für Literale im GAST (I) 43
3.19 Beispiel für Literale im GAST (II) 44
3.20 Initialisierung einer Liste 44
3.21 Beispiel einer Funktionsdeklaration 47
3.22 Beispiel eines Funktionsaufrufs 48
3.23 Beispiel einer Paketdeklaration 54
3.24 Beispiel einer Attributdeklaration 55
3.25 Beispiel für eine Deklaration einer Operation 56
3.26 Beispiel für eine Konstruktordeklaration 56
3.27 Beispiel für eine Aufzählung im GAST 58
3.28 Beispiel für eine generische Schnittstelle im GAST 60
3.29 Beispiel für eine generische Klasse im GAST 64
4.1 Alternativen zu den elementweisen Operatoren **und** und **oder** . . 69
4.2 klassische C Enum 72
4.3 Umsetzung einer C-Enum in GAST 72

4.4 Beispiele für Casts in Java 73
5.1 Beispiel zum instanceof Fehler 82
5.2 Eine lokale Klasse 83
5.3 Eine anonyme Klasse 83
5.4 Eine lokale Klasse mit innerer Klasse 83
5.5 qualifizierte Angabe der inneren Klasse 84
5.6 Ein lokaler Feldzugriff 84
5.7 Ein nicht-lokaler Feldzugriff 85
5.8 Das XML Stylesheet zur Erzeugung von Java Quellcode 86
7.1 Beispiel für eine problematische innere Klasse 97

Tabellenverzeichnis

3.1 Analyse zum GAST Umfang 16
3.2 IDs der Basisdatenstrukturen 28

8.1 Beschreibung der Kommandozeilenparameter von `GAST2Java.sh` . 109
8.2 Beschreibung der Kommandozeilenparameter der Skripte zum Erstellen der `XML` Dateien 110

9.1 Werte zum Projekt HelloWorld 115
9.2 Werte zum Projekt CodeCollection 116

Abbildungsverzeichnis

1.1 Ein möglicher Parsebaum für `i = i + 1` 4
1.2 Ein abstrakter Syntaxbaum 5

2.1 Enumeration für den Typ der binären Operatoren 11
2.2 Klassendiagramm zum Element *expression* 12
2.3 Klassendiagramm zum Element *binaryExpression* 12
2.4 Klassendiagramm zum Element *statement* 13
2.5 Klassendiagramm zum Element *localFunctionDeclaration* 13
2.6 Klassendiagramm zum Element *repeatUntilLoop* 13
2.7 Klassendiagramm zum Konflikt der *namelist* 14

3.1 Struktur eines GAST-Dokuments 18
3.2 Externe Basistypen im GAST 19
3.3 Externe Basistypen im GAST 20
3.4 Details zur Typreferenz im GAST 22
3.5 Einbettung der Basisdatenstrukturen in die Typhierarchie 23
3.6 Funktionalität der Basisdatenstruktur Array 23
3.7 Funktionalität der Basisdatenstruktur List 25
3.8 Funktionalität der Basisdatenstruktur Map 27
3.9 Modell für ein Label im GAST 29
3.10 Modell für den Block im GAST 30
3.11 Modell für eine Variablendeklaration im GAST 30
3.12 Modell für eine Anweisung zur Deklarieren von Methoden 32
3.13 Modell für die Ausdrucksanweisung des GASTs 32
3.14 Modell für die Zweifachauswahl des GASTs 33
3.15 Modell für die Fallauswahl des GASTs 35
3.16 Modell für die zählende Schleife des GASTs 36
3.17 Modell für die interne Deklaration einer Variablen 36
3.18 Modell für die iterierende Schleife des GASTs 37
3.19 Modell für die bedingungsgesteuerte Schleife des GASTs 38
3.20 Modell für eine Anweisung zum Auslösen von Fehlern im GAST . 39
3.21 Modell für überwachte Anweisungen im GAST 40
3.22 Modell für die Return Anweisung im GAST 40

3.23 Modell für die Break Anweisung im GAST 41
3.24 Modell für die Break Anweisung im GAST 42
3.25 Modell für Ausdrücke im GAST 43
3.26 Modell für ein Literal im GAST 44
3.27 Der GAST für eine Zuweisung an eine Variable 45
3.28 Modell für einen Zugriff auf eine Variable im GAST 45
3.29 Modell für einen unären Ausdruck im GAST 45
3.30 Modell für einen binären Ausdruck im GAST 46
3.31 Modell für den ternären Ausdruck im GAST 46
3.32 Modell für eine Funktionsdefinition im GAST 47
3.33 Modell für einen Funktionsaufruf im GAST 48
3.34 Modell für Programme im GAST 53
3.35 Modell für Pakete im GAST 54
3.36 Die verschiedenen Typdeklarationen des GASTs 55
3.37 Deklaration von Feldern im GAST 56
3.38 Zugriffe auf Felder im GAST 57
3.39 Details zur Aufzählungsdeklaration im GAST 58
3.40 Details zur Deklaration komplexer Typen im GAST 59
3.41 Objektdiagramm zur Umsetzung von <T> 60
3.42 Objektdiagramm zur Umsetzung von <T extends A> 60
3.43 Objektdiagramm zur Umsetzung von <int a> 61
3.44 Objektdiagramm zur Umsetzung einer Ausprägung mit <SomeType> 61
3.45 Objektdiagramm zur Umsetzung einer Ausprägung mit <1024> . 62
3.46 Details zum Aufruf eines Superkonstruktors 62
3.47 Details zur Deklaration von Klassen im GAST 63
3.48 Details zu den Modifiern im GAST 63
3.49 Details zu Sichtbarkeiten im GAST 64
3.50 Sprachspezifische Fragmente im GAST (I) 65
3.51 Sprachspezifische Fragmente im GAST (II) 66

4.1 Umsetzung des Casts aus Listing 4.4 Zeile 3 74

5.1 Sequenzdiagramm zum Buildvorgang (1) - Vorbereitungen 80
5.2 Sequenzdiagramm zum Buildvorgang (2) - Parsevorgang 81
5.3 Das Interface **ITransformer** 81

6.1 Baum zum Ausdruck (4+3)*2 90

7.1 Schritte zur Erstellung des GASTs aus JAST-Instanzen 93
7.2 Objektdiagramm zur Umsetzung von `int[]` im GAST 100
7.3 Objektdiagramm für einen Präfixoperator im GAST 101
7.4 Objektdiagramm für einen Postfixoperator im GAST 101

8.1 Öffnen der Datei `plugin.xml` 106
8.2 Starten des Export Wizards 106
8.3 Optionen des JaML Plug-ins 108
8.4 Icon zum Starten des Plug-ins 108
8.5 Dialog zur Auswahl des Zielprojekts 109
8.6 Optionsdialog für GAST2Java 109

9.1 Chart zum Projekt HelloWorld 114
9.2 Chart zum Projekt HelloWorld 116
9.3 Chart zum Projekt CodeCollection 117
AnhangLiteraturverzeichnis

Literaturverzeichnis

[1] Assoziation (uml). online. http://de.wikipedia.org/wiki/Assoziation_(UML).

[2] Eclipse downloads home. http://www.eclipse.org/downloads/.

[3] Java online documentation: Interface node. online. http://java.sun.com/j2se/1.5.0/docs/api/org/w3c/dom/Node.html.

[4] Java se downloads. http://java.sun.com/javase/downloads/index.jsp.

[5] Unreachable code. online. http://en.wikipedia.org/wiki/Dead_code.

[6] Xslt tutorial. online. http://www.w3schools.com/xsl/.

[7] Mda guide version 1.0.1. online, June 2003. http://www.omg.org/docs/omg/03-06-01.pdf#search=%22OMG%20mda%20guide%2%2.

[8] Alfred V. Aho, Ravi Sethi, and Jeffry D. Ullman. *Compilerbau Band 1*. ADDISON-WESLEY, 2nd edition, 1992.

[9] Alfred V. Aho, Ravi Sethi, and Jeffry D. Ullman. *Compilerbau Band 2*. ADDISON-WESLEY, 2nd edition, 1992.

[10] Paul Anderson. The performance penalty of xml for program intermediate representations. http://www.dcs.kcl.ac.uk/staff/mark/scam2005/anderson-XML.pdf.

[11] Azad Bolour. Notes on the eclipse plug-in architecture. July 2003. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.

[12] Gegor Fischer and Jürgen Wolff von Gudenberg. *Programmieren in Java 1.5*. Xpert.press. Springer Verlag, 2005.

[13] Gregor Fischer and Jürgen Wolff von Gudenberg. Improving the quality of programming education by online assessment. In Gitzel et al. [14], pages 208–211.

[14] Ralf Gitzel, Markus Aleksy, Martin Schader, and Chandra Krintz, editors. *Proceedings of the 4th International Conference on Principles and Practices of Programming in Java, Mannheim, Germany, August 30 – September 1, 2006*, volume ?? of *ACM International Conference Proceedings*. Mannheim University Press, 08 2006.

[15] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. ADDISON-WESLEY, 3rd edition, 2005.

[16] Prof. Dr. Peter A. Henning and Prof. Dr. Holger Vogelsang, editors. *Taschenbuch Programmiersprachen*. Fachbuchverlag Leipzig, 2004.

[17] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. *Lua 5.0 Reference Manual*. Tecgref - Computer Sience Department - PUC-Rio, November 2003. `www.lua.org`.

[18] Joachim Lusiardi. Bugreport: [1.5][compiler] instanceof accepts primitive type as left-hand-side. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=137918`.

[19] Joachim Lusiardi. Bugreport: Ast - instanceof - getlength returns wrong length. online. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=128077`.

[20] Joachim Lusiardi. Bugreport: Ast: errors with parameter array and full qualified types. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=128960`.

[21] Joachim Lusiardi. Bugreport: Ast: errors with parentheses expressions in for-init initialisers. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=128961`.

[22] OMG. Model driven architecture specification. Version 1.0, May 2003 via `http://www.omg.org`.

[23] OMG. Unified Modeling Language specification, 2003. Version 2.0, June 2003 via `http://www.omg.org`.

[24] Roland Petrasch and Oliver Meimberg. *Model Driven Architecture Eine praxisorientierte Einführung in die MDA*. dpunkt.verlag, 1. auflage edition, 2006.

[25] JDT/Core team. Eclipse java development tools subproject. Master's thesis. `http://www.eclipse.org/jdt/`.

[26] Inc. The MathWorks. Short-circuit operators. online. `http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f0-38948.html#logical_shortcircuit`.

[27] Doug Tidwell. *XSLT*. O'Reilly, November 2001.

[28] Eric van der Vlist. *XML Schema*. O'Reilly, first edition edition, June 2002.

[29] Wikipedia. Backus-naur-form. Wikipedia. http://de.wikipedia.org/wiki/Backus-Naur-Form.

[30] Wikipedia. Lua programming language. Wikipedia. http://www.lua.org.

[31] Wikipedia. Nassi-shneiderman-diagramm. Wikipedia. http://de.wikipedia.org/wiki/Nassi-Shneiderman-Diagramm.

Danksagungen

Mein Dank gilt all jenen, die durch ihre fachliche und seelische Unterstützung diese Arbeit überhaupt erst ermöglicht haben. Besonderen Dank gebührt folgenden Personen:

Professor Dr. Jürgen Wolff von Gudenberg für seine äußerst zuvorkommende Betreuung während dieser Diplomarbeit. Häufiges Lesen und Korrigieren der Ausarbeitung dürfte einige Fehler und Inkonsistenzen frühzeitig identifiziert haben.

Dipl. Inf. Gregor Fischer für seine kompetenten Ratschläge und Hilfestellung zu allen Fragen aus dem Bereich XML und den verwandten Techniken sowie der stetigen psychologischen Unterstützung.

Meiner Freundin **Jittiporn Chaisaingmongkol** und meinem Vater **Gerhard Lusiardi**, die mir beide viel Verständnis entgegenbrachten und nie aufgehört haben an mich zu glauben.

Marco Nehmeier für seine Unterstützung beim Entwurf der Algorithmen aus Kapitel 4.

Andrea Kaufmann für ihre vielen Stunden, die sie mit einer ihr unverständlichen Materie beim Korrekturlesen verbracht hat.

Allen Mitarbeitern des Lehrstuhl 2 und allen Hiwis, die mich während der Diplomarbeit durch ihre Toleranz und Hilfsbereitschaft unterstützt haben.

Last but not least **Fritz Kleemann** für die hervorragende Pflege des Lehrstuhlnetzes und seine Hinweise bei diversen Shellskripten.

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Diplomarbeit selbständig und ohne unzulässige, fremde Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Die bildlichen Darstellungen, Tabellen, Quelltexte und Graphen habe ich selbst angefertigt.

Würzburg, den 29. September 2006