Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Arne Bernin

Entwicklung eines Compilers für eine C-ähnliche Programmiersprache für den LEGO Mindstorms NXT

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Arne Bernin

Entwicklung eines Compilers für eine C-ähnliche Programmiersprache für den LEGO Mindstorms NXT

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Gunter Klemke
Zweitgutachter : Prof. Dr. rer. nat. Kai von Luck

Abgegeben am 24. August 2006

Arne Bernin

Thema der Bachelorarbeit

Entwicklung eines Compilers für eine C-ähnliche Programmiersprache für den LEGO Mindstorms NXT

Stichworte

LEGO NXT, Compiler, NQC

Kurzzusammenfassung

Ziel dieser Arbeit ist die Entwicklung eines Compilers. Dieser soll eine textbasierte Programmiersprache für das LEGO Mindstorms NXT Robot Kit übersetzen, dabei orientiert sich die Sprache grob an der Syntax der Programmiersprache C. Diese Arbeit bietet eine Übersicht der verwendeten Werkzeuge, der entstandenen Sprache und der aufgetretenen Probleme. Dabei werden die einzelnen Aspekte der Compilererstellung unter Java berücksichtigt.

Arne Bernin

Title of the paper

Development of a Compiler for a C-like programming language for the LEGO Mindstorms NXT

Keywords

LEGO NXT, Compiler, NQC

Abstract

This paper describes the development of a compiler for a C-like programming language for the LEGO NXT Robotic Kit. It gives an overview of the theoretical background, the used tools, the generated language and the problems that occured during the development of the software. The paper thereby discusses the aspects of compiler development in Java.

Ole Kirk Christiansen, dem Begründer von LEGO, der mir schon vor den Robotern interessante Stunden schenkte.

Danksagung

Danke an John Hansen für die Entwicklung von NXT Byte Code.

Danke an Gunter und Kai für Kritik, Anregungen und Unterstützung.

Ich danke allen, die mich während des Schreibens dieser Arbeit unterstützt haben, besonders Conny für die unermüdlichen Korrekturen.

Inhaltsverzeichnis

Tabellenverzeichnis 10

Abbildungsverzeichnis 11

1. Einführung 12
- 1.1. Motivation . . . 13
- 1.2. Zielsetzung und Problemstellung . . . 14
- 1.3. Einordnung der Arbeit . . . 14
- 1.4. Aufbau der Arbeit . . . 15
- 1.5. Begriffe . . . 15
- 1.6. Schreibweisen . . . 15

2. Analyse 16
- 2.1. Anforderungen an die Sprache . . . 16
- 2.2. Anforderungen der Zielgruppe . . . 17
- 2.3. Ähnliche Sprachen . . . 17
- 2.4. Das Zielsystem . . . 18
  - 2.4.1. Einführung . . . 18
  - 2.4.2. Informationsbeschaffung . . . 18
  - 2.4.3. Hardware . . . 18
  - 2.4.4. Vergleich mit LEGO Mindstorms RCX . . . 20
  - 2.4.5. Betriebssystem/Firmware . . . 20
  - 2.4.6. Alternative Betriebssysteme . . . 21
- 2.5. LabView Entwicklungsumgebung . . . 21
- 2.6. NBC . . . 22
- 2.7. NBC als Zielsprache . . . 23
- 2.8. Compiler . . . 24
  - 2.8.1. Einführung . . . 24
  - 2.8.2. Architektur eines Compilers . . . 24
  - 2.8.3. Benutzung von NQC als Basis . . . 25
  - 2.8.4. Benutzung von GCC . . . 25
  - 2.8.5. Entwicklung eines eigenen Compilers . . . 25
  - 2.8.6. Verwendete Programmiersprache . . . 26
  - 2.8.7. Komplette Eigenentwicklung . . . 26

2.8.8. Metacompiler 26
2.8.9. Wahl des Metacompilers 28

3. Design **30**
3.1. Namensgebung 30
3.2. Einfluss von JavaCC und JTB 30
3.2.1. Syntaxbaum 30
3.2.2. Visitor(en) 31
3.2.3. Semantische Analyse 34
3.2.4. Symboltabelle 34
3.3. Architektur 34
3.3.1. Frontend 35
3.3.2. Backend 35
3.3.3. Zwischencode 36
3.4. Gemeinsamkeiten zu NQC 36
3.5. Unterschiede zu NQC 36
3.6. Einschränkungen von NNQC 37
3.7. Ein Beispiel 38
3.8. Grammatik 39
3.8.1. Notation 39
3.8.2. EBNF 40
3.8.3. Nachweis der LL(k) Eigenschaften 42
3.8.4. Kontextfreie Grammatiken und Programmiersprachen 45
3.8.5. Typische Probleme 45
3.8.6. Dangling-Else-Problem 45
3.9. Präprozessor 46
3.9.1. Funktionsumfang 46
3.9.2. Verwendete Werkzeuge 46
3.9.3. Grammatik des Präprozessors 46
3.9.4. Fehlerbehandlung 47
3.9.5. Include-Zyklen 47
3.9.6. Zeilennummernproblematik 48
3.10. Zwischencode 50
3.11. Aufruf des NBC Compilers 51
3.12. Fehlerbehandlung 51
3.13. Bibliotheksfunktionen 51
3.14. Funktionen, Tasks und Subroutinen 52
3.15. Lokale Variablen 53
3.16. Ablaufsteuerung 53
3.17. Codegenerierung 53
3.17.1. Direkte Übersetzung von Statements 54
3.17.2. Tasks und Subroutinen 54

3.17.3. Funktionsaufrufe . . . 54
3.17.4. Repeat-Schleifen . . . 55
3.17.5. Do-while und while-do-Schleifen . . . 56
3.17.6. Zuweisungen . . . 56
3.17.7. If-then-else . . . 56
3.17.8. switch-case . . . 56
3.17.9. Arithmetische Ausdrücke . . . 57
3.17.10. Vergleichsausdrücke . . . 57

4. Realisierung **58**
4.1. Ablaufsteuerung . . . 58
4.2. Präprozessor . . . 58
4.2.1. Aufruf des Präprozessors . . . 58
4.2.2. Include-Zyklen . . . 58
4.3. Compiler . . . 59
4.3.1. Probleme mit der Fehlerbehandlung . . . 59
4.3.2. Visitoren . . . 59
4.3.3. Codegenerierung . . . 60
4.3.4. Aufruf von NBC . . . 61
4.3.5. Unterstütze Kommandozeilenparameter . . . 62
4.4. Tests . . . 62
4.5. Standardbibliothek . . . 62
4.5.1. print . . . 63
4.5.2. get_ultra_sensor(int value&) . . . 63
4.5.3. start_motor(int motor, int speed) . . . 63
4.6. Erweiterungsmöglichkeiten . . . 63
4.6.1. Eclipse Plugin . . . 63
4.6.2. Codeoptimierung und Zwischencode . . . 64
4.6.3. Erweiterungen der Sprache . . . 64
4.6.4. Direkte Ausgabe von Bytecode . . . 65
4.7. Bewertung . . . 65

5. Schluss **67**
5.1. Rückblick und Bewertung . . . 67
5.2. Fazit . . . 68

Literaturverzeichnis **69**

A. Quellcode **72**
A.1. NNQC BNF Grammatik . . . 72
A.2. nnqc.jtb . . . 74
A.3. precompiler.jtb . . . 80

A.4. Pseudocode für die Zielcodeerzeugung 83
A.4.1. Do-while und while-do-Schleifen 83
A.4.2. Zuweisungen 83
A.4.3. If-then-else 84
A.4.4. switch-case 84

B. NNQC-Beispiele **86**
B.1. Helloworld 86
B.2. Playtone 86

Glossar **88**

Tabellenverzeichnis

3.1. Unterschied zwischen NQC und NNQC 38
3.2. Auszug aus der Parsetabelle 43

4.1. Kommandozeilenparameter 62

Abbildungsverzeichnis

1.1. Lego Nxt AlphaRex 12

2.1. Lego Nxt 19
2.2. Lego Labview 22

3.1. Ein Syntaxbaum 31
3.2. Sequenzdiagramm Visitor 32
3.3. Die Schichten der Architektur 35
3.4. Visitor(en) und Globale Daten 37
3.5. Zyklenerkennung beim include von Dateien 48
3.6. Include von Dateien 49

4.1. Visitor Klassenhierarchie 59

1. Einführung

Roboter erobern nicht nur den Weltraum[1], Vorgärten[2] und Wohnzimmer[3], sie machen auch vor den Kinderzimmern nicht halt. Die Firma LEGO hat bereits 1999 mit ihrem Mindstorms Robotic Kit (ab hier mit RCX abgekürzt) für Furore gesorgt und dies nicht nur im Kinderzimmer. Auch im Bereich der Ausbildung in Schulen und Universitäten erfreuen sich die LEGO-basierten Roboter großer Beliebtheit. Ganze Ligen dieser Roboter spielen Fußball gegeneinander oder retten in unbekanntem Gelände in Not geratene Plastikmännchen. Mit dem Erscheinen des LEGO Mindstorms NXT (NXT) schickt sich LEGO nun an, diesen Erfolg fortzusetzen.

Abbildung 1.1.: Lego Nxt AlphaRex , Quelle: [ALPHAREX (2006)]

[1] Jules Verne, das neue autonome Versorgungsverhikel der ESA [JVERNE (2003)]
[2] als autonome Rasenmäher wie RoboMower [ROBOMOWER (2006)]
[3] wie Roomba, ein autonomer Staubsauger [ROOMBA (2006)]

Grundsätzlich gibt es zwei Möglichkeiten eine Steuerung von mobilen Robotern zu realisieren. Die zur Steuerung verwendete Software kann auf einem externen System ablaufen und die nötigen Daten – beispielsweise per Funk – übermitteln. Auf dem Roboter läuft dabei nur eine Minimalsoftware, die Befehle an die Motoren weiterleitet und Sensordaten zum Steuerungssystem übermittelt[4]. Die andere Möglichkeit ist das Ausführen der Steuerungssoftware auf dem Roboter selber. Die Entwicklung der Software läuft dabei getrennt vom Laufzeitsystem ab, die eingesetzte Entwicklungsumgebung erzeugt einen Binärcode, der auf das Zielsystem übertragen und dort ausgeführt wird. Dadurch wird ein autonomes Agieren und Reagieren auf Sensordaten möglich. Für die RCX-Serie kam dieser Ansatz zum Tragen, der per Crosscompiler erzeugte Code wurde auf den Roboter übertragen.
Um die LEGO Mindstorms herum enstand eine Community, die eine ganze Reihe von Hard- und Software-Erweiterungen wie Integrierte Entwicklungsumgebungen (IDE), zusätzliche Sensoren oder alternative Betriebssysteme hervorbrachte. Die deutlich leistungsfähigere Hardware sowie die neuen Sensoren des LEGO Mindstorms NXT verbessern die Möglichkeiten noch einmal. Der Ansatz des NXT als autonomes System bildet deshalb die Grundlage dieser Arbeit.

1.1. Motivation

Roboter wie die der Mindstorms-Serie bieten sich an, um jungen Menschen die Informatik als Sympathieträger näher zu bringen. Sie bieten einen spielerischen Einstieg in die Welt des Programmierens, die Ergebnisse und Auswirkungen sind sofort und konkret sichtbar.

Die für den RCX vorhandenen Erweiterungen und Entwicklungsumgebungen machen den Einsatz im Bereich der Lehre an Schulen und Universitäten möglich, um Konzepte der Programmierung und des Agierens autonomer Roboter zu verdeutlichen. Insbesondere die Verfügbarkeit von klassischen – textbasierten – Programmiersprachen ist dafür eine notwendige Voraussetzung, da sie die Gelegenheit zum Einstieg in die Art der Programmierung bietet, die Standard im Bereich der Softwareentwicklung ist. Diese Anforderungen kann die von LEGO zur Verfügung gestellte grafische Programmierung nicht erfüllen. Eine einfache, textbasierte Programmiersprache ist für den NXT derzeit nicht verfügbar, was die Einsatzmöglichkeit stark beschränkt. Diese Arbeit soll einen Beitrag dazu leisten, diese Lücke zu schließen.

[4] Nach diesem Konzept arbeitet das Robotics Studio von Microsoft [MSROBOTICS (2006)]. Ein Windows-PC wird benutzt, um die Roboter per Bluetooth oder WLAN zu steuern.

1.2. Zielsetzung und Problemstellung

Die Entwicklung von Software für autonome Systeme erfolgt (wie bei anderen eingeschränkten Systemen auch) zumeist nicht auf dem System selber, sondern in einer davon unabhängigen Entwicklungsumgebung. Das fertige Programm wird mit einem Crosscompiler für das Zielsystem übersetzt und dann auf dieses übertragen. Der übliche Zyklus von Programmentwicklung, Test und Fehlerbehebung wird durch die Trennung von Entwicklungs- und Zielsystem erschwert, ein direktes Debugging erfordert besondere Unterstützung durch das Zielsystem.
Ziel dieser Arbeit ist die Entwicklung eines Prototyps für einen Crosscompiler für den LEGO Mindstorms NXT. Dieser soll es ermöglichen, mit einer syntaktisch an C angelehnten Sprache Programme zu schreiben und diese auf dem LEGO Mindstorms NXT ablaufen zu lassen. Dabei müssen die Anforderungen des geplanten Einsatzgebietes im Bereich des Informatikunterrichts an Schulen berücksichtigt werden.
Der Compiler soll über die Kommadozeile aufgerufen werden, eine grafische Oberfläche ist nicht vorgesehen. Der Compilerlauf beginnt mit der Übergabe der Quelldateien über Kommandozeilenparameter und endet mit der Ausgabe des Bytecodes in eine, ebenfalls per Parameter spezifierte, Datei.
Als Programmierumgebung kann jeder übliche Texteditor verwendet werden. Die Erstellung einer Erweiterung für eine Integrierte Entwicklungsumgebung (IDE) ist nicht Teil dieser Arbeit, sie soll jedoch die Voraussetzung schaffen, dass dies im Rahmen einer weiteren Arbeit erfolgen kann.

1.3. Einordnung der Arbeit

Diese Arbeit ist im Bereich der Angewandten Informatik anzusiedeln. Sie basiert auf den Grundlagen des Compilerbaus, die einen Querschnitt durch wichtige Teile der Informatik darstellen. Dazu gehören Formale Sprachen – insbesondere Programmiersprachen – Algorithmen und Software-Engeneering.
Compiler und die zugehörigen Programmiersprachen bilden das Fundament der praktischen, angewandten Informatik. Sie bilden die Schnittstelle zwischen dem Menschen als Programmierer und der Maschine in Form des Prozessors, ohne sie wäre die Ausführung von Programmen in Form von Software[5] nicht möglich. Durch die große Relevanz ergibt sich der Vorteil, das zahlreiche Veröffentlichungen über die Entwicklung von Programmiersprachen und Compilern verfügbar sind, wie auch zahlreiche Werkzeuge zur Unterstützung des Compilerbaus.

[5] Also als austauschbares, nicht “fest-verdrahtetes” Programm im Prozessor.

1.4. Aufbau der Arbeit

Im folgenden Kapitel werden die Bedingungen und Voraussetzungen analysiert, die die Grundlagen dieser Arbeit bilden. Dies beinhaltet eine Übersicht der NXT-Hardware und der verschiedenen Möglichkeiten zum Erstellen eines Compilers. Darauf aufbauend erfolgt im dritten Kapitel die Entwicklung eines Designs für die Zielsprache sowie den eigentlichen Compiler. Die folgende Umsetzung des Designs wird im vierten Kapitel näher betrachtet, dort wird auch ein Blick auf die aufgetretenen Probleme und künftigen Erweiterungsmöglichkeiten geworfen. Die Arbeit schließt mit einem Rückblick auf den eingeschlagenen Weg sowie einem Fazit ab.

1.5. Begriffe

Die Ausdrücke Betriebssystem und Firmware in Bezug auf den LEGO Mindstorms NXT werden in dieser Arbeit synonym verwendet. Eine Übersicht über weitere Abkürzungen und Begriffe findet sich im Glossar (ab Seite 90).

1.6. Schreibweisen

Zur Hervorhebung von besonderen Worten und Begriffen wird die folgende Notation verwendet:

- **Variablen- und Objekttypen** sind durch Fettdruck gekennzeichnet.
- *Methodenaufrufe, Programmcode oder Funktionen* sind durch Schrägschrift erkennbar.
- Verweise auf Einträge in der Literaturliste (die ebenso für die Angabe von Webseiten verwendet wird) sind in eckigen Klammern [] zu finden.

2. Analyse

In diesem Kapitel werden die Resultate der Analyse dargestellt. Ziel ist dabei sowohl die genauere Definition des zu erreichenden Ziels, als auch die Darstellung der Bedingungen, Fragen und Probleme, die damit zusammenhängen. Die hier gewonnenen Kenntnisse fließen als Grundlage in die spätere Phase des Designs ein.
Im konkreten Fall sollen die folgenden Fragen betrachtet werden:

- Wie lässt sich die C-ähnliche Zielsprache genauer definieren?
- Wie beeinflusst die Zielgruppe der zukünftigen Anwender die Sprache?
- Welche ähnlichen Sprachen gibt es, besonders im Bereich LEGO Mindstorms? Lassen sie sich als Basis nutzen?
- Gibt es bereits bestehende Ansätze in diesem Bereich ? Was für bestehende Software läßt sich nutzen ?
- Welche relevanten Eigenschaften hat die Hardware, welche die Firmware?
- Welche Werkzeuge stehen für den Compilerbau zur Verfügung, welches ist das Richtige in diesem Fall?
- Sind schon jetzt Probleme absehbar? Manche Probleme treten erst in der Designphase wirklich zu Tage.

2.1. Anforderungen an die Sprache

Die entwickelte/implementierte Programmiersprache soll keine allgemeingültige Programmiersprache (wie etwa Java/C/Perl) werden, sie dient dem primären Zweck, einen LEGO Mindstorms NXT zu programmieren. Die Rücksichtnahme auf Portierbarkeit ist diesem untergeordnet. Eine Ähnlichkeit zu bestehenden Sprachen ist aber durchaus sinnvoll, um beispielsweise den Umstieg zu erleichtern oder auf bestehende Informationen über ähnliche Sprachen zurückgreifen zu können. Das schließt natürlich auch die Weiterverwendung vorhandener Dokumentation ein.
Die Sprache selber muss in einer eindeutigen Form definiert werden (wie z. B. mit Hilfe einer Grammatik in EBNF). Dies ist sowohl für spätere BenutzerInnen nötig, als auch

für die Implementierung bzw. die Benutzung von Werkzeugen zur Erstellung des Compilers.

2.2. Anforderungen der Zielgruppe

Aus der Tatsache, dass die Sprache im Informatikunterricht der Oberstufe an Schulen eingesetzt werden soll, ergeben sich natürlich weitere Anforderungen. Insbesondere die Einfachheit der Syntax spielt dabei eine wichtige Rolle, die Sprache muss leicht verständlich und erlernbar sein. Dies bedingt den Verzicht auf komplizierte Konstrukte wie Zeigerarithmetik. Die Vermittlung wichtiger Programmierkonzepte (wie etwa Kapselung, Iterationen, etc.) soll möglich sein.
Für den Einsatz an Universitäten wären die nötigen Einschränkungen deutlich weiter gefasst, da hier komplexe Programmiersprachen wie Java oder C Teil der Ausbildung sind. Eine einfach zu verwendende Programmiersprache ist hier allerdings auch von Vorteil.
Im Vergleich zur grafischen Programmierung mit Blocksymbolen, wie sie etwa die LabView Programmierumgebung (siehe 2.5) bietet, sind die Möglichkeiten der textbasierten Programmierung universeller. Grafische Programmierung bietet durch die direkete Visualisierung zwar einen einfachen Einstieg in die Programmierung, insbesondere für Kinder ist dies vorteilhaft. Auf dem weiteren Weg zur komplexeren Programmierung tritt die Notwendigkeit der direkten Visualisierung jedoch weiter in den Hintergrund.

2.3. Ähnliche Sprachen

Neben der Sprache C selbst[1], existiert für die Mindstorms RCX ein weiterer Vertreter der C-ähnlichen Sprache genannt 'Not Quite C' kurz NQC [NQC (2005)]. Diese stellt zusammen mit Java für die leJos Firmware die am meisten verwendete, textbasierte Programmiersprache für die alten Mindstorms Roboter dar. Sie unterliegt allerdings recht stark den Beschränkungen der Firmware, ist dafür aber mit der Standardfirmware von LEGO direkt einsetzbar. Der NQC-Compiler ist in C++ geschrieben. Aufgrund seines ähnlichen Einsatzgebietes und des relativ geringen Sprachumfanges bietet sich NQC sehr gut als Basis für die Entwicklung einer eigenen, modifizierten Version an, die den erweiterten Funktionen des LEGO Mindstorms NXT gerecht wird. NQC verfügt über einen integrierten Präprozessor, der die Sprache um Makros, Definitionen und includes erweitert. Über Makros wird ein großer Teil der nötigen Funktionalität, beispielsweise zur Steuerung der Motoren, abgedeckt.

[1] Die allerdings nicht mit der Standard Firmware nutzbar ist.

2.4. Das Zielsystem

2.4.1. Einführung

Beim vorgesehenen Zielsystem handelt es sich um das Mindstorms NXT Robotic Kit von LEGO A/S. Dieses wird voraussichtlich ab September 2006 im Handel verfügbar sein. Derzeit gibt es lediglich Vorabversionen für Beta-Tester. Die Hochschule für Angewandte Wissenschaften Hamburg nimmt an diesem Beta-Test-Programm teil und verfügt über einen LEGO Mindstorms NXT.

2.4.2. Informationsbeschaffung

Die Tatsache, dass das Zielsystem derzeit nicht im Handel erhältlich ist, hat massive Auswirkungen auf die Verfügbarkeit von Informationen. Die Hochschule für Angewandte Wissenschaften Hamburg verfügt über ein Testsystem, das ihr im Rahmen des Mindstorms Developer Program (MDP) zur Verfügung gestellt wurde. Das MDP ermöglicht Zugriff auf ein internes Forum, das der Kommunikation mit anderen Entwicklern dient. Außerdem bietet es Zugriff auf (einige) interne Dokumente von LEGO. Die Dokumentation ist zum gegenwärtigen Zeitpunkt noch recht lückenhaft. Aus anderen Quellen ist wenig Zusätzliches zu finden, insbesondere weil am LEGO Mindstorms NXT Interessierte oft keinen Zugriff auf die Hardware[2] haben. Deshalb ist die Gesamtmenge an verfügbaren Informationen derzeit eher gering.

2.4.3. Hardware

Die zentrale Steuerungseinheit des LEGO Mindstorms NXT besteht aus einem viereckigen Baustein mit Kunststoffgehäuse, das über ein internes Batterie-/Akkufach, ein grafisches LCD Display sowie über drei Anschlüsse zum Ansteuern der Motoren als auch vier Anschlüsse zum Verbinden von Sensoren verfügt. Die groben technischen Spezifikationen[3] sind:

- 32-bit ARM7 microcontroller (genauer ein Atmel AT91SAM7S256, 48 Mhz) mit
- 256 Kbytes FLASH, 64 Kbytes RAM
- 8-bit AVR Mikrokontroller[4] (Atmel ATmega48) mit
- 4 Kbytes FLASH, 512 Byte RAM

[2] Die Gesamtmenge der außerhalb von LEGO kursierenden Systeme dürfte derzeit unter 200 liegen
[3] Quelle: [NXTSPEC (2006)]
[4] Dieser Koprozessor dient zur Verarbeitung des Ein- und Ausgabe von Motoren und Sensoren.

Abbildung 2.1.: Lego Nxt, Quelle: [LEGONXT (2006)]

- Bluetooth wireless communication (Bluetooth Class II V2.0 compliant, USB Bluecore 4 v2.0+EDR)
- USB full speed port (12 Mbit/s)
- 4 input ports, jeweils 6 adriges Kabel (analog/digital) (Ein Port mit IEC 61158 Type 4/EN 50 170 kompatibeler RS-485 Schnittstelle)
- 3 output ports, jeweils 6 adriges Kabel (analog/digital)
- 100 x 64 pixel LCD Grafik Display
- Lautsprecher - 8 kHz Sound Qualität. Sound Kanal mit 8-bit Auflösung and 2-16 KHz sample rate.
- Stromquelle: 6 AA Batterien oder Akkus

Mitgelieferte Sensoren

Das komplette LEGO Mindstorms NXT Kit verfügt über vier unterschiedliche Sensoren:

1. **Lichtsensor:** Dieser Sensor ist analog angebunden und liefert einen Integer-Wert zwischen 0 und 1023 , abhängig vom Umgebungslicht.
2. **Schalter:** Digitaler Sensor, auch als Bumper benutzbar.
3. **Ultraschallsensor:** Angebunden über den I^2C-Bus, erlaubt die Erkennung von Hindernissen, und Entfernungsmessungen.

4. **Mikrofon:** analog, liefert einen Integer-Wert zwischen 0 und 1023. Dieser ist abhängig von der umgebenden Lautstärke, die vom Mikrofon registriert wird.

Erweiterungsmöglichkeiten

Die eingeschränkte Zahl an analogen und digitalen Eingängen lässt sich durch die Benutzung des integrierten I^2C-Busses und der auf Port 4 vorhandenen seriellen RS-485 Schnittstelle erhöhen. Verschiedene Hersteller haben die baldige Verfügbarkeit von zusätzlichen Sensoren (etwa der Digital Kompass von Hitechnic [HITECHNIC (2006)]) angekündigt. Mit einem I/O Multiplexer wie lepomux [LEPOMUX (2006)] für den alten RCX ist sicherlich auch in naher Zukunft zu rechnen. Der Betrieb von alten Sensoren der RCX Baureihe ist prinzipiell möglich, es wird dafür von LEGO voraussichtlich ein Adapterset geben, um einen Anschluss der inkompatiblen Stecker zu ermöglichen.
Die Stecker zum Anschluss der Motoren und Sensoren an den LEGO Mindstorms NXT sind leider (wieder) LEGO Eigenentwicklungen, in diesem Fall keine Stecker im “LEGO-Stein-Format", sondern spezielle Ausführungen von RJ-12 Steckern[5].

2.4.4. Vergleich mit LEGO Mindstorms RCX

Die von LEGO für den LEGO Mindstorms NXT verwendete Hardware ist bedeutend leistungsfähiger als die der alten RCX-Serie. In dieser kam ein 8-bit Mikrokontroller (Hitachi H8) mit 32KB Ram und 16KB Rom zum Einsatz. Im Gegensatz zur alten Hardware die nur über eine Infrarotschnittstelle verfügte, kommt zur drahtlosen Übertragung nun Bluetooth zum Einsatz. Als Ersatz für die alte serielle Schnittstelle verfügt der LEGO Mindstorms NXT über USB 2.0.
Auch die Motoren wurden verbessert, so sind Sensoren zur Zustandmessung (ebenso wie das Getriebe) integriert worden und lassen sich auch durch den NXT auslesen[6]. Insgesamt lässt sich festhalten, dass die neue Hardware deutlich vielseitiger ist, was die technischen Fähigkeiten angeht. Sie entfernt sich jedoch - schon durch die Optik - vom klassischen LEGO Design. Dies setzt sich in den neuen Steckern und dem nichtmodularen Aufbau der Motoren mit integriertem Getriebe fort.

2.4.5. Betriebssystem/Firmware

LEGO liefert auf dem NXT eine vorinstallierte Firmware mit, die sich über die von LEGO mitgelieferte Software aktualisieren lässt. Wie schon bei den älteren Mindstorms Generatio-

[5] RJ12 6X6 DEC MMJ Modular Plug
[6] So lässt sich kontrollieren, ob ein Motor blockiert ist. Ebenso lässt sich der derzeitige Winkel wie auch die Anzahl der Umdrehungen ermitteln.

nen, ermöglicht die Firmware nur das Ausführen von Programmen, die in einem speziellen Bytecodeformat vorliegen. Die Ausführung von Programmen, die im Maschinencode des verwendeten ARM7-Prozessors vorliegen, ist nicht möglich. Die Dokumentation des Bytecodeformats ist derzeit nicht zugänglich.
Aus dem Compiler heraus den Bytecode zu erzeugen, ist also nicht einfach möglich, ein aufwendiges Reverse-Engineering zur Analyse des Bytecodeformats wäre nötig. Da die Dokumentation aber in den nächsten Monaten mit dem Sourcecode freigegeben wird, wäre dies weitgehend vergebene Mühe.
LEGO hat angekündigt, den Sourcecode der Firmware unter einer Open-Source-Lizenz freizugeben, der genaue Zeitpunkt steht allerdings noch nicht fest.

2.4.6. Alternative Betriebssysteme

Zum Zeitpunkt des Entstehens dieser Arbeit gibt es keine alternativen Firmware-Implementierungen für den NXT. Im Gegensatz dazu existiert für die alte RCX-Serie eine Reihe von unterschiedlichen Betriebssystemen (Lejos, BrickOS). Ein derzeitiger Hinderungsgrund für alternative Implementierungen ist die zu ungenaue Dokumentation der verwendeten Spezialbausteine und der Spezifikation der Verbindungen untereinander (Bus, Protokoll, eventuelle Watchdogs). Die über das Mindstorms Developer Program verfügbaren Informationen sind zwar präziser, doch hat nicht jeder Zugriff darauf und die Dokumentation dort weist noch erhebliche Lücken auf. Zudem ist die Verbreitung der Hardware auf die Beta-Tester beschränkt. Die fehlenden Informationen sollen allerdings von LEGO zu gegebener Zeit veröffentlicht werden, spätestens mit der Freigabe des Sourcecodes der NXT Firmware werden die nötigen Details natürlich bekannt sein.
Die weitere Verbreitung der Hardware nach dem offiziellen Verkaufsstart dürfte die Chancen für die Entwicklung einer alternativen Firmware allerdings deutlich steigern.
Als Zielsystem bleibt die Standard-Firmware als derzeit einzig existierendes Betriebssystem für den LEGO Mindstorms NXT festzuhalten.

2.5. LabView Entwicklungsumgebung

Die von LEGO selbst vorgesehene Entwicklungsumgebung zum Erstellen von Software für den NXT basiert auf LabView von National Instruments. In dieser werden Programme in einer grafischen Darstellung zusammengefügt. Es gibt Blocksymbole für unterschiedliche Sprachkonstrukte wie Schleifen oder Sensordaten. Diese haben – je nach Typ – zusätzliche Einstellmöglichkeiten wie die Konfiguration von Wiederholungen oder das Warten auf einen bestimmten Wert. Die LabView Software ist für den geplanten Compiler nicht nutzbar, da

es keine API gibt, auf der man aufsetzen könnte. Eine Darstellung der LabView Umgebung ist in Abbildung 2.2 zu sehen.

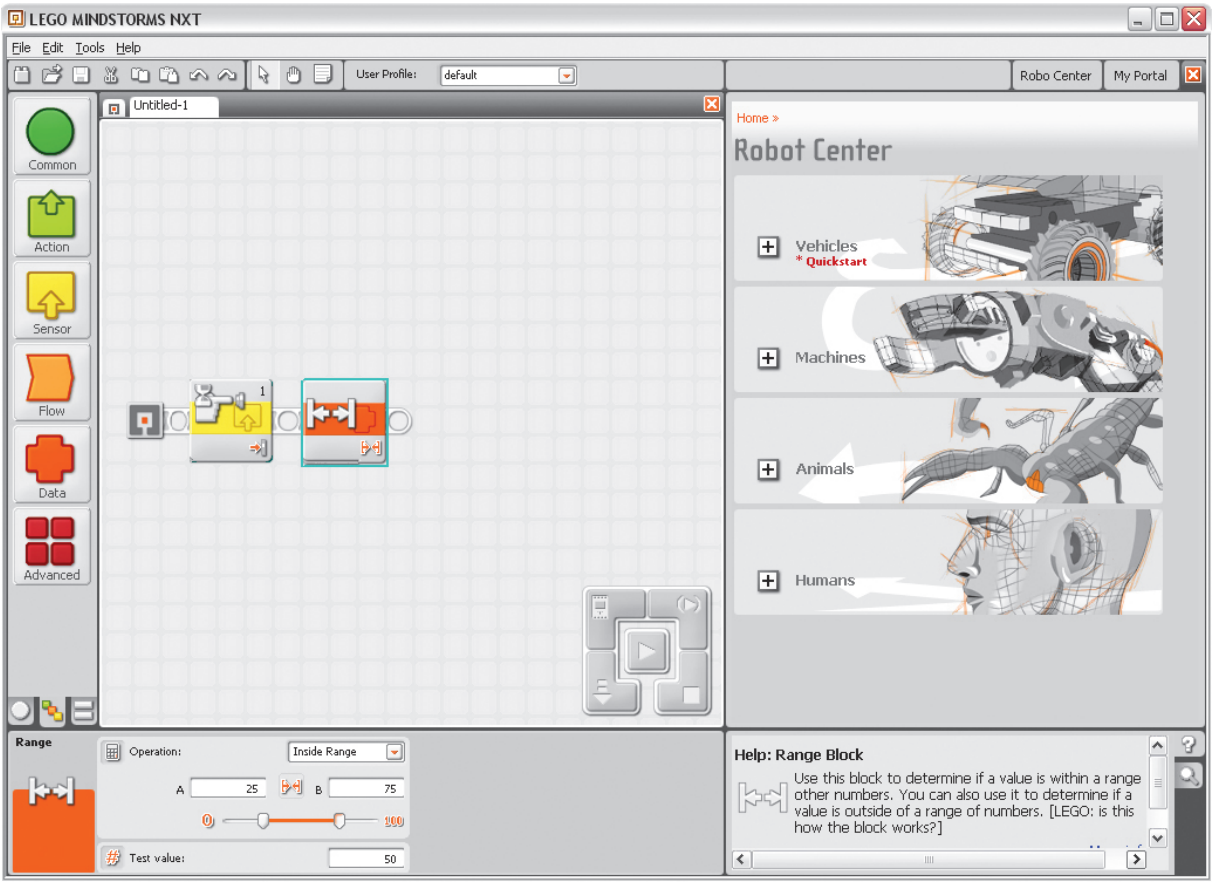

Abbildung 2.2.: Lego Labview

2.6. NBC

NXT Byte Code (NBC) ist zum Zeitpunkt des Schreibens dieser Arbeit die einzige textbasierte Programmiersprache für den LEGO NXT. Entwickelt wurde sie von John Hansen[7]. NBC orientiert sich an den Möglichkeiten der LEGO Mindstorms NXT Firmware und der Umsetzung in Bytecode. Die Sprache ist Assembler nachempfunden, auch wenn sie einige Besonderheiten aufweist:

- So werden natürlich Sprunglabels unterstützt, aber auch die direkte Benutzung von

[7]Dem Entwickler von brixcxx (einer Entwicklungsumgebung für den LEGO RCX) und NQC

Subroutinen und Threads[8]. Es gibt allerdings keinen integrierten Mechanismus für die Parameterübergabe.

- Es gibt keine Register (der Bytecode wird von der Firmware ausgeführt, nicht direkt vom Prozessor).
- Direkte arithmetische Ausdrücke ohne Variablen können direkt verwendet werden, ein *set x , 3+4*7* führt zum gewünschten Ergebnis (x wird auf 31 gesetzt).
- Es gibt keinen direkten Speicherzugriff auf Adressen, der Zugriff ist nur über symbolische Variablennamen möglich. Eine Ausnahme bildet hier der Zugriff auf Ein- und Ausgaben der Sensoren, die per vordefinierten Adressen direkt angesprochen werden (IO-Mapped).
- Eine Zeigerarithmetik ist nicht vorhanden.
- Code kann nicht über Adressen angesprungen werden.
- Es gibt keinen Stack auf dem lokale Variablen oder Rücksprungadressen abgelegt werden können.
- Es gibt Strukturen und Strings, die direkt unterstützt werden.
- Die Kommunikation mit dem Betriebssystem erfolgt über Systemaufrufe (*syscall*).

NXT Byte Code ist in Delphi geschrieben, und als Binärprogramm für die Betriebssysteme Windows und Mac OS X unter der MPL[9] verfügbar. Der Sourcecode ist derzeit nicht veröffentlicht.
Die Einschränkung der unterstützten Plattformen lässt sich mit wine [wine (2006)] umgehen, dies ist für viele (UNIX-) Plattformen verfügbar. Das Kommandozeilentool NBC bietet neben dem Kompilieren und Übertragen der Dateien auf den LEGO Mindstorms NXT außerdem die Möglichkeit, Bytecode zu dekompilieren und ihn in Form von NBC Sourcecode auszugeben. Dies lässt sich beispielsweise nutzen, um die mit LabView generierten Programme zu analysieren.

2.7. NBC als Zielsprache

Die derzeit einzige Möglichkeit, außerhalb von LabView ausführbaren Code für den LEGO Mindstorms NXT zu erzeugen, liegt in der Benutzung von NBC. Der zu erstellende Compiler muss also NBC als Zielcode ausgeben und danach den NBC Compiler benutzen,

[8] Vergleichbar mit den Tasks in NQC
[9] Mozilla Public Licence

um diesen in Bytecode umzuwandeln. Der Compiler ist somit ein Transcompiler, er übersetzt von einer Sprache in eine andere (und nicht direkt in Maschinencode beziehungsweise Bytecode).

2.8. Compiler

2.8.1. Einführung

> Im Grunde ist ein Compiler ein Programm, das ein in einer bestimmten Sprache – der Quell-Sprache – geschriebenes Programm liest und es in ein aquivalentes Programm einer anderen Sprache, der Ziel-Sprache übersetzt. Eine wichtige Teilaufgabe des Compilers besteht darin, dem Benutzer Fehler, die im Quellprogramm enthalten sind, zu melden.
>
> [DRACHENBUCH (1999)] ,Seite 1

An dieser Stelle soll neben dem obigen Zitat nur eine kurze Übersicht über die Komponenten eines Compilers gegeben werden. Für tiefergehende Betrachtungen sei hier auf die einschlägige Literatur ([GUETING (1999)], [DRACHENBUCH (1999)], [WIRTH (1995)]) beziehungsweise entsprechende Online-Einführungen ([HELMICH (2006)], [VOELLER (2006)], [WPCOMPILERBAU (2006)]) als Einstieg verwiesen.

2.8.2. Architektur eines Compilers

Eine häufig verwendete Architektur bei Compilern ist die Zweiteilung in Frontend und Backend. Dabei besteht die Aufgabe des Frontends darin, den einzulesenden Quelltext zu zerlegen, zu analysieren, und bei Fehlerfreiheit, in einen Zwischencode zu überführen. Das Backend erzeugt aus diesem Zwischencode dann den endgültigen Zielcode in einer anderen Sprache oder als Binär-, beziehungsweise Bytecode. Der Zwischencode soll also unabhängig von beidem sein[10].
Diese Trennung dient dazu sowohl die Quellsprache als auch die Zielsprache relativ einfach wechseln zu können, sofern die entsprechende Schnittstelle (der Zwischencode) korrekt genutzt wird.
Während das Frontend meist mit Werkzeugen aus einer Grammatik automatisch generiert wird, ist beim Backend eher “Handarbeit” nötig.
Kurz gefasst besteht das Frontend (meist) aus einem Scanner und einem Parser. Der Scanner

[10] Im Falle der GNU Compiler Collection GCC ist dies beispielsweise für alle Quellsprachen (C, Java, C++) eine Sprache namens Register Transfer Language (RTL).

zerlegt die vorgefundenen Strings im Quelltext anhand von regulären Ausdrücken und übergibt diese zur syntaktischen Prüfung an den Parser. Dieser generiert dann schließlich den Zwischencode, sofern auch zusätzliche semantische Überprüfungen keine Fehler entdeckt haben.

2.8.3. Benutzung von NQC als Basis

Die direkte Benutzung des NQC-Sourcecodes als Basis für die Entwicklung des Compilers scheitert aus mehreren Gründen. Zum einen an den fehlenden Kenntnissen der benutzten Sprache (C++), die eine umfangreiche Einarbeitung voraussetzen würde. Zum anderen erzeugt der NQC Compiler Bytecode, der zudem noch für den Mindstorms RCX optimiert und somit nicht kompatibel ist. Dafür wird keine unabhängige Zwischensprache benutzt, sondern der Bytecode direkt generiert. Der Zuschnitt auf den Umfang der alten Firmware macht zudem eine Veränderung der Sprache notwendig.
Das alles zusammengenommen lässt den Aufwand für das Umschreiben der NQC nicht geringer erscheinen, als einen Compiler auf Basis der Sprache NQC selber zu entwickeln.

2.8.4. Benutzung von GCC

Eine alternative Möglichkeit für die Erstellung des Compilers wäre die Benutzung einer bestehenden Software wie beispielsweise die GNU Compiler Collection (GCC). Nach Durchsicht der verfügbaren, umfangreichen Dokumentation wird diese Möglichkeit aus folgenden Gründen verworfen. Zum Einen müssten sowohl ein neues Frontend als auch ein Backend geschrieben werden. Zum Anderen ist das System zur Erzeugung von Maschinencode für reale Prozessoren ausgelegt, d.h. es wird beispielsweise das Vorhandensein von Registern erwartet. Zudem liegt der Umfang des Sourcecodes im Bereich von etlichen MegaByte, der Einarbeitungsaufwand wäre hoch. Durch die Verwendung von GCC würden die Vorteile der Benutzung von Java entfallen.

2.8.5. Entwicklung eines eigenen Compilers

Die Entwicklung eines eigenen Compilers ist eine mögliche Alternative zur Veränderung eines bestehenden. Der Aufwand scheint nicht größer zu sein, im Gegenteil eher geringer. Insbesondere, da eine umfangreiche Einarbeitung in die Programmiersprache oder Software entfällt, und – je nach verwendeten Werkzeugen – auch auf schon vorhandenes Wissen aus dem Bereich Compilerbau zurückgegriffen werden kann.

2.8.6. Verwendete Programmiersprache

Als Programmiersprache für den Compiler soll Java verwendet werden. Dies hat mehrere Gründe. Zum Einen garantiert die Plattformunabhängigkeit von Java die Lauffähigkeit auf allen wichtigen Plattformen, ohne auf spezifische Eigenheiten des darunterliegenden Betriebssystems bedeutend Rücksicht nehmen zu müssen. Zum Anderen ermöglicht die Verwendung von Java die nahtlose Integration in die derzeit am meisten verbreitete Entwicklungsumgebung Eclipse. Dabei können Teile des Compilers (Scanner und Parser) wiederverwendet werden.
Metacompiler (siehe 2.8.8), die sich für die schnelle Entwicklung von Compilern eignen, sind für Java verfügbar.
Ein Grund, der gegen den Einsatz von Java beim Compilerbau angeführt werden kann, ist die gegenüber C vorhandene geringere Ausführgeschwindigkeit des Java-Bytecodes und der Overhead der Objektorientierung. Dadurch ergibt sich eine längere Laufzeit des Übersetzungsvorgangs. Bei einem Compiler spielt dies allerdings nur dann eine Rolle, wenn dadurch das Maß des Erträglichen – aus Sicht der Anwender – überschritten wird. Dies hängt eher von den benutzten Algorithmen und vom Umfang des zu übersetzenden Programms ab, als der Ausführgeschwindigkeit der verwendeten Programmiersprache. Die zu erwartende Größe der NNQC-Programme im Bereich von mehreren KiloBytes lässt keine großen Probleme im Bereich der Compiler-Laufzeit erwarten.
Unter diesen Voraussetzungen ist die Verwendung von Java für den Compiler eine legitime Wahl.

2.8.7. Komplette Eigenentwicklung

Einen Compiler komplett “von Hand” zu entwickeln, ist unnötig aufwendig und fehlerträchtig. Es gibt zur Erzeugung sehr ausgereifte Compiler-Compiler (Metacompiler), die zumindest die Generierung des Scanners und des Parsers übernehmen.

2.8.8. Metacompiler

Metacompiler dienen der Erzeugung von Compilern oder Teilen von ihnen. Ihr Funktionsumfang unterscheidet sich durchaus beträchtlich, ebenso wie die von ihnen geforderten Voraussetzungen, was insbesondere den Aufbau der Grammatik der Quellsprache angeht. Dadurch bedingt unterscheiden sich auch die Konzepte des Parsings, es gibt sowohl Metacompiler die eine Top-Down-Analyse verwenden, als auch andere, die auf Bottom-Up setzen. Beides hat natürlich Vor- und Nachteile. Im Folgenden werden einige der weiter verbreiteten Werkzeuge für die Compilergenerierung in Java vorgestellt. Dabei handelt es sich in der Regel um Scanner- bzw. Parsergeneratoren.

Lex/Yacc

Lex und Yacc, bzw. die zahlreichen Derivate und Nachbauten dieser Programme, sind die Klassiker, wenn es um die Erzeugung von Compilern geht. Die meisten der in C geschriebenen Compiler sind mit Hilfe dieser Werkzeuge erstellt worden, der GNU C Compiler ist hier ein berühmtes Beispiel. Lex dient zur Erzeugung eines Scanners, der dann zusammen mit dem – von Yacc erzeugten – Parser benutzt wird. Die von Yacc erwartete Grammatik muss vom Typ LALR(1) sein. Für Lex existieren unterschiedliche Implementierungen in Java, beispielsweise JLex [JLEX (2006)] und JFLex [JFLEX (2006)]. Eine Yacc-Implementierung in Java ist beispielsweise CUP [CUP (2006)].
Es ist allerdings auch möglich – wenn gleich nicht ratsam – nur den mit Lex generierten Scanner zu benutzen, und den Parser selber zu implementieren. Dies bedeutet einen unnötig erhöhten Aufwand.

Jay

Jay (siehe [JAY (2006)]) ist ein in C geschriebener Parsergenerator, der die entsprechenden Klassen in Java generiert. Er basiert auf BSD Yacc und verwendet LR(1) Grammatiken. Jay unterscheidet sich nicht grundlegend von Yacc.

JavaCC

JavaCC, der Java Compiler Compiler ist ein Metacompiler, der ursprünglich von der Firma Suntest im Jahre 1995 entwickelt wurde. Der Besitzer der Rechte an JavaCC, bzw. die Firma die die Weiterentwicklung vorantrieb, wechselte noch drei weitere Male. Die Software landete schließlich bei den Sunlabs von Sun Microsystems und – der bis dahin nicht verfügbare – Sourcecode wurde schließlich freigegeben. Die Besonderheit von JavaCC ist die Verwendung von LL(k) Grammatiken für den Scanner/Parser, der erzeugte Parser funktioniert nach der Methode des rekursiven Abstiegs. Es gibt für JavaCC Präprozessoren, die die automatische Generierung eines Konkreten beziehungsweise Abstrakten Syntaxbaumes (AST) und der diesen repräsentierenden Klassen ermöglichen. JavaCC und der Sourcecode sind frei verfügbar und stehen unter der BSD Lizenz [BSDLIC (1998)]. Die in der Version 4.0 vorliegende Software macht einen ausgereiften Eindruck, sie wird inzwischen selbst benutzt, um Scanner und Parser für JavaCC zu generieren. JavaCC erwartet eine Java 1.5/5.0 kompatible JRE.
Mehr Informationen zu JavaCC finden sich auf der JavaCC-Homepage [JAVACC (2006)].
Für JavaCC ist ein Plugin für Eclipse erhältlich [JAVACCECLIPSE (2006)], dieses bietet Syntaxhighlighting für JavaCC Grammatiken und den automatischen Aufruf von JavaCC. Es unterstützt ebenfalls JJTree und JTB.

JJTree und JTB

JJTree und JTB sind Erweiterungen für JavaCC, die den erzeugten Compiler um die nötigen Klassen zur Erzeugung eines AST erweitern. Dabei verfolgen sie unterschiedliche Ansätze. JTB nimmt für sich in Anspruch besonders einfach und simpel in der Bedienung zu sein, während JJTree mehr Aufwand erfordert, dafür aber auch flexibler ist. Der große Vorteil von JTB gegenüber JJTree besteht darin, dass bei JTB keine Änderungen an der – ursprünglich für JavaCC entwickelten – Grammatik nötig sind. Im Gegensatz dazu sind für JJTree massive Anpassungen notwendig. JJTree ist in der JavaCC Distribution enthalten, auch die Dokumentation findet sich bei JavaCC [JJTREEDOC (2006)]. JTB ist extern unter der BSD Lizenz verfügbar [JTB (2006)]. Eine gute Übersicht über die beiden Erweiterungen findet sich unter der Quelle [JTBJJTREE (2006)] . JTB 1.3 erwartet als Mindestanforderung eine Java 1.5/5.0 kompatibles JRE.

Weitere Metacompiler

Es gibt eine Vielzahl von Compilergeneratoren für Java, oft entstanden aus unterschiedlichen Projekten an Hochschulen. Zudem einige wenige, (ursprünglich) kommerziell entwickelte wie JavaCC. Im WWW finden sich detaillierte Übersichten (z.B. [JAVACOMPILER (2006)]).

2.8.9. Wahl des Metacompilers

Bei der Wahl des zu benutzenden Metacompilers sind einige Kriterien zu beachten:

1. Die Software soll ausgereift sein, am besten schon länger entwickelt werden.
2. Es soll sichergestellt sein, das sie auch in Zukunft weiterentwickelt wird oder zumindest Support/Bugfixes verfügbar sind (Mailinglisten, Foren).
3. Die Software muss – wie auch der Sourcecode – unter einer freien Lizenz verfügbar sein.
4. Eine Ausreichende Dokumentation ist online verfügbar.
5. Kurze Einarbeitungszeit.
6. Die Verfügbarkeit einer größeren Anzahl von Beispielgrammatiken ist sinnvoll.

JavaCC erfüllt mit JTB zusammen alle diese Kriterien. Insbesondere die Verwendung von LL(k) Grammatiken in Verbindung mit rekursivem Abstieg sind dem Autor aus der Vorlesung “Compiler- und Interpreter” gut bekannt. Durch die Verwendung von JTB sind keine wesentlichen Änderungen an der Grammatik nötig[11]. Dies sollte sich vorteilhaft auf die Entwicklungsgeschwindigkeit auswirken.

[11] allerdings, wie sich noch zeigen wird, manchmal sinnvoll

3. Design

Dieses Kapitel bietet eine Überblick über das geplante Design des Compilers. Dabei werden die einzelnen Komponenten der Software und ihre Funktion näher betrachtet. Das erstellte Konzept wird später in der Phase der Realisierung umgesetzt.

3.1. Namensgebung

Die angestrebte Nähe zu NQC legt es nahe, dieses ebenfalls im Namen der Programmiersprache deutlich zu machen. Eine Vielzahl von Projekten tut dies durch das Hinzufügen von einzelnen Worten oder dem Aufbau von Akronymen (beispielsweise GNU, GNU is not UNIX), die einen Hinweis auf den Ursprung geben. Dieser Tradition folgend, wird die hier entwickelte Sprache im weiteren Verlauf als NNQC ("Nearly Not Quite C" oder auch "NXT Not Quite C") bezeichnet werden.

3.2. Einfluss von JavaCC und JTB

Die verwendeten Programmierwerkzeuge JavaCC und JTB haben erheblichen Einfluss auf das Design des Compilers, sowohl auf die zu verwendende Grammatik, als auch auf die Art des Zwischencodes bzw. die Erzeugung des fertigen NBC-Codes.

3.2.1. Syntaxbaum

Die von JavaCC und JTB generierten Klassen erzeugen beim Parsen einen Abstrakten Syntaxbaum[1] (AST). Abbildung 3.1 zeigt einen beispielhaften, vereinfachten Syntaxbaum nach dem Parsen einer Quelldatei. **Input** ist dabei die Wurzel des Baumes, sie wird später auch vom Parser als Einstiegspunkt für Visitoren (vergleiche 3.2.2) zurückgeliefert. In diesem

[1] Da die Möglichkeit besteht JavaCC mitzuteilen, welche Tokens vom Scanner zu erkennen, aber nicht vom Parser zu beachten sind, werden diese nicht in den Baum eingefügt. Somit enthält der Baum nur die wichtigsten Token, er entspricht nicht der vollständigen Ableitung und ist insofern kein Konkreter, sondern ein Abstrakter Syntaxbaum.

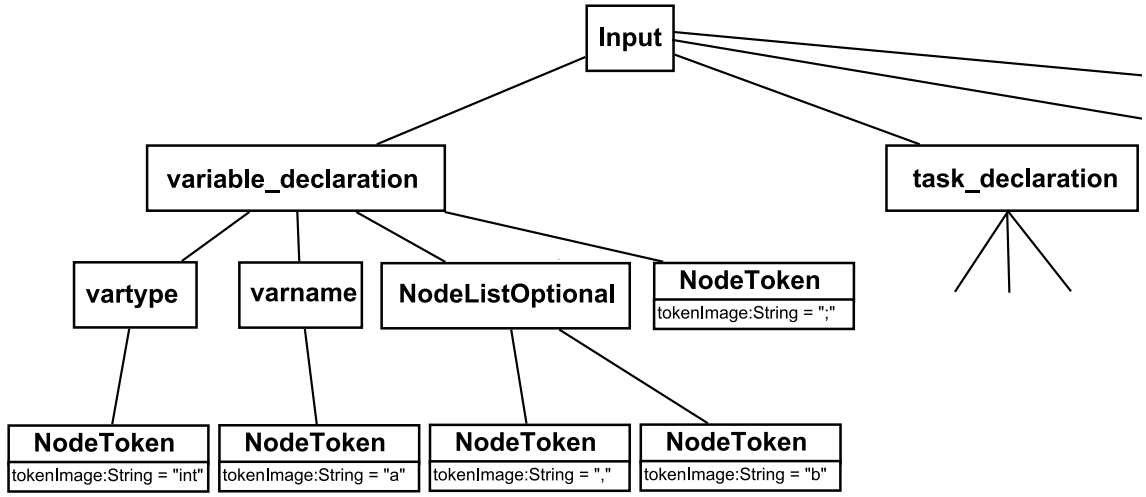

Abbildung 3.1.: Ein Syntaxbaum

Beispiel werden zwei globale Variablen vom Typ **Integer** (*int a,b;*) deklariert. Die Speicherung der einzelnen Strings des Quellcodes erfolgt in Objekten vom Typ **NodeToken**, das Attribut **tokenImage** ist dabei die Stringrepräsentation des durch den Scanner erkannten Tokens. Alle Knoten im Syntaxbaum implementieren das Interface **Node**.
NodeListOptional wird benutzt, wenn es - wie der Name schon sagt - eine optionale Liste von Elementen des Typs **Node** (bzw. **NodeToken**[2]) gibt. In diesem Fall sind nur zwei Objekte für eine zusätzliche **int** Deklaration enthalten (*","* und *"b"*), die Liste könnte natürlich auch mehr Nodes enthalten oder leer sein.
Der generierte Syntaxbaum ist typisiert, d.h. die Knoten im Baum sind beispielsweise vom Typ **variable_declaration** oder **varname**. Ein Umbau des Baumes zur direkten Umwandlung in einen unabhängigen Zwischencode wird durch die Typisierung erschwert, da Änderungen in der Struktur verhindert werden (Es werden keine einfache Knoten als Kinder etc. erwartet, sondern Knoten eines bestimmten Typs). Es wäre jedoch möglich, einen eigenen Baum parallel aufzubauen, der die veränderte Struktur abbildet. Dieses ist zur weiteren Analyse des Codes und der späteren Codeerzeugung jedoch nicht zwingend notwendig.

3.2.2. Visitor(en)

Zur Traversierung des vom Parser erzeugten Syntaxbaumes, werden von JTB mehrere Klassen bereitgestellt. Diese beruhen auf dem Visitor-Design-Pattern. Bei diesem werden die einzelnen Knoten des Baumes durchlaufen und je nach vorgefundenem Typ des Knotens, unterschiedlicher Code ausgeführt. Dies wird durch eine Anzahl von unterschiedlichen Methodensignaturen für die Methode *visit* realisiert, die als Argument einen Typ aus dem Syn-

[2]Nodes und NodeTokens sind in diesem Falle äquivalent, da ein Blatt im Baum immer ein NodeToken ist.

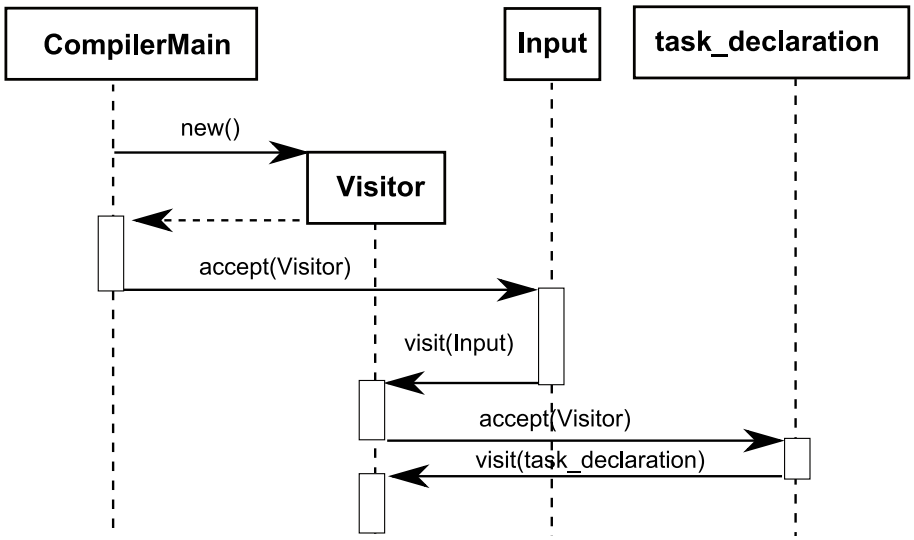

Abbildung 3.2.: Sequenzdiagramm Visitor

taxbaum erwarten, also beispielsweise **statement**. Beispielhaft ist dies in Abbildung 3.2 dargestellt:

1. **CompilerMain** erzeugt über *new* ein neues Objekt vom Typ **Visitor** (Visitor ist eigentlich nur das Interface, aber zur Demonstration sollte dies ausreichend sein).
2. **CompilerMain** ruft die *accept* Methode von **Input** auf und übergibt den zuvor erzeugten **Visitor** als Parameter. **Input** ist der Typ des Wurzelknotens des AST. Durch den Aufruf von *visit* in der Wurzel beginnt die Traversierung des Baumes.
3. **Input** ruft nun seinerseits die Methode *visit* des Visitors auf, und übergibt eine Referenz auf sich selbst als Parameter. Alle weiteren Knoten des Baumes, die über ein *accept* angesprochen werden, verhalten sich ebenso: Sie übergeben immer eine Referenz auf sich selbst als Argument.
4. Die Methode *visit* des Visitors ruft nun wiederum ein *accept* auf, nämlich das von **task_declaration**. Die Referenz auf **task_declaration** bekommt der Visitor über die *getChild* Methode von Input, die von allen Knoten unterstützt wird. Je nach Visitor wird hier eventuell auch Code ausgeführt, der mehr als nur das Traversieren des Baumes zur Folge hat. Der generierten **DepthFirstVisitor** beschränkt sich darauf, alle Kinder eines Knotens über *accept* anzusprechen.
5. **task_declaration** ruft seinerseits wieder die *visit* Methode des Visitors auf.
6. Das Szenario wiederholt sich, bis der Baum traversiert ist.

Das Entscheidende an diesem “ping-pong” der gegenseitigen Aufrufe ist die automatische Auswahl, welche *visit* Methode im Visitor aufgerufen wird. Diese Entscheidung wird

anhand der Methodensignatur von der Java-VM getroffen, die Benutzung von Abfragen (wie z. B. per “if") ist so überflüssig. Durch die unterschiedlichen *visit* Methoden ist der Programmcode für die einzelnen Knoten zudem gut gegliedert.
Der als Standard von JTB generierte **DepthFirstVisitor** bietet Zugriff auf die Blätter **NodeTokens** des gesamten Baumes in genau der Reihenfolge, in der auch der Sourcecode geparst wurde (Tiefentraversierung). Alle weiteren Visitoren sollen von **DepthFirstVisitor** erben. Die vorhandenen Methoden des **DepthFirstVisitors** traversieren den Baum lediglich. Soll nun bei einem bestimmten Knoten (Argument-Typ im Methodenaufruf) etwas anderes als die weitere, reine Traversierung passieren, so wird die entsprechende Methode in der Unterklasse überschrieben.
Über die Visitoren wird der gesamte Zugriff auf den Syntaxbaum geregelt, sie sind die “Arbeitspferde” für per JTB/JavaCC generierte Compiler.
Zur Erfüllung der einzelnen Aufgaben, werden unterschiedliche Visitoren benötigt, die jeweils den Baum traversieren und über das **CompilerGlobals** Objekt Zugriff (siehe 3.3.3) auf alle nötigen Daten (Variablen, Tasks, Funktionen, . . .) haben. Diese Daten werden dann von den einzelnen Visitoren erweitert.

Für die folgende Funktionalität sollen Visitoren implementiert werden:

- **Definition:** Zusammensuchen der Definition von Strukturen, Tasks, Subroutinen und Funktionen.
- **Deklaration:** Sammeln der Deklaration von lokalen und globalen Variablen, Überprüfen des Vorhandenseins der entsprechenden Definition bei Strukturen.
- **Semantik:** Überprüfen der Einhaltung semantischer Regeln, beispielsweise die Überprüfung des Typs bei Zuweisungen oder das Vorhandensein eines Tasks mit dem Namen “main”. Überprüfung ob eine Variable vor der Benutzung korrekt deklariert und initialisiert wurde.
- **CodeCollector:** Einfügen der Codeblöcke mit dem eigentlichen Programmcode in die entsprechenden Daten-Objekte (Funktion, Subroutine, Task). Diese Blöcke werden als Liste von Objekten des Typs **statement** gespeichert.

Über den Weg des **CompilerGlobals** Objektes sind danach alle Daten verfügbar, um die Generierung des Zielcodes vorzunehmen.
Die Benutzung von Exceptions innerhalb von *visit* Methoden ist nicht möglich, da das Werfen von Exceptions entsprechende Änderungen (nämlich “throws ParseException”) in den *visit* Methoden der Oberklasse **DepthFirstVisitor** erfordern würde, also das (manuelle) Umschreiben der generierten Klassen. Diese Änderungen wären bei einer Neugenerierung unweigerlich verloren und müßten von Hand gepflegt werden. Zusätzlich wird durch die Exception die Traversierung des Baumes unterbrochen. Aus diesen Gründen ist für die Fehlerbehandlung in den Visitoren ein anderer Weg zu nehmen (siehe Abschnitt 3.12).

3.2.3. Semantische Analyse

Eine der Möglichkeiten, im Compilerbau mit semantischen Überprüfungen umzugehen, ist die Benutzung einer Attributierten Grammatik. JavaCC bietet keine direkte Syntax zum Definieren einer solchen Grammatik. Allerdings gibt es die Möglichkeit, Java-Code zur Semantischen Analyse hinzuzufügen, der direkt vom Parser ausgeführt wird. Unglücklicherweise verbietet JTB die Benutzung dieser Funktionalität, da es (als Präprozessor) selber darauf zurückgreift, um die Erstellung des Syntaxbaums beim Parsen durchzuführen.
Die für JTB vorhandene Alternative ist die Verwendung eines Visitors für die semantische Analyse (siehe 3.2.2).

3.2.4. Symboltabelle

Beim Compilerbau ist es üblich, Informationen über Tokens, beziehungsweise Symbole in einer Symboltabelle zu sammeln. Dabei werden im Normalfall einige Symbole (von festen Sprachelementen wie Schlüsselwörtern) beim Start des Programms in diese Tabelle geschrieben. Der Scanner fügt dann weitere hinzu (z. B. Variablen), der Parser ergänzt eventuell (beispielsweise den Typ), so dass die Informationen schließlich immer umfangreicher werden.
Die von JavaCC/JTB generierten Scanner und Parser verwenden keine Symboltabelle. Sie erzeugen für gleiche Strings auch jedesmal wieder ein neues **Token**, da dieses die Informationen über sein Auftreten im Quellcode (Spalte, Zeile) enthält. Da die Benutzung von JTB das Einbetten von Code in der Grammatik ausschließt, ist ein spezieller Visitor die einzige Möglichkeit, nachträglich eine Symboltabelle zu erzeugen.
Statt eine allgemeine Symboltabelle zu verwenden, ist es ebenso möglich, die Informationen in speziellen, getrennten Datenstrukturen (beispielsweise über ein assoziatives Array (Hashmap) zum schnellen Zugriff) zu speichern. Dazu werden die Informationen in getrennten Objekten für Variablen, Strukturen, Tasks, Funktionen und Subroutinen gespeichert.
Diese Variante kommt für den NNQC-Compiler zum Einsatz.

3.3. Architektur

Wie schon in der Analyse erwähnt (siehe 2.8.2), besteht die Grobarchitektur eines Compilers meist aus einem Frontend und einem Backend, das über unabhängigen Zwischencode das Programm erhält.
Diese Struktur lässt sich auch in dem Design dieses Compilers wiederfinden. Eine Übersicht über die einzelnen Schichten findet sich in Abbildung 3.3.

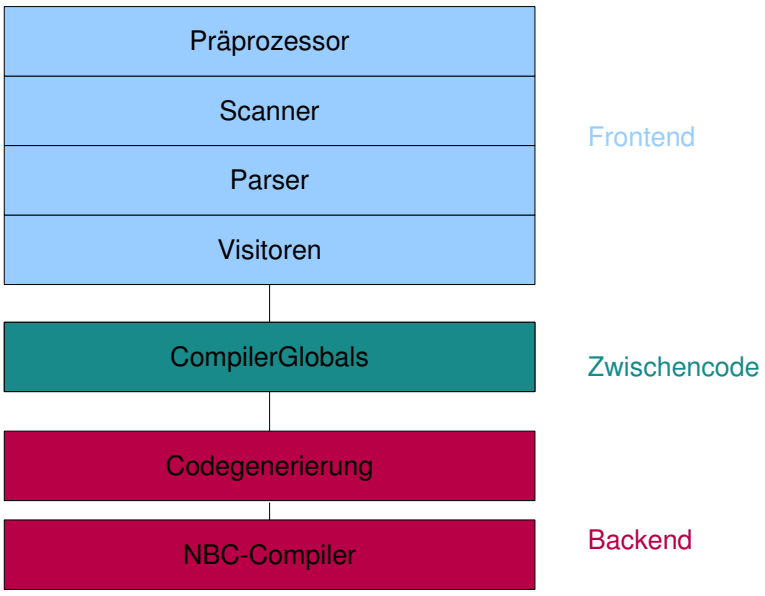

Abbildung 3.3.: Die Schichten der Architektur

3.3.1. Frontend

Das Frontend verarbeitet den Quellcode. Die einzelnen Schritte lassen sich dabei als Schichten darstellen. Wie Abbildung 3.3 zu entnehmen ist, besteht das Frontend des Compilers aus vier Schichten:

- **Präprozessor:** Dieser liest die Dateien ein und fügt sie zu einem Ganzen zusammen, zur Übergabe an den Scanner.
- **Scanner:** Zerlegt den Programmtext in Zeichenketten (Tokens) zur Übergabe an den Parser.
- **Parser:** Dieser führt die eigentliche syntaktische Analyse aus, und speichert das Ergebnis im Syntaxbaum.
- **Visitoren:** Diese traversieren den Syntaxbaum und verarbeiten die dort vorhandenen Informationen. Die dabei anfallenden Daten werden in der folgenden Schicht (Zwischencode) gespeichert.

3.3.2. Backend

Das Backend erzeugt aus dem Zwischencode den fertigen Zielcode. Im Falle des NNQC-Compilers besteht es aus den folgenden Komponenten:

- **Codegenerierung:** Benutzt die darüber liegende Schicht zum Zugriff auf die Daten, erzeugt eine Ausgabedatei im NBC Format.
- **NBC-Compiler:** Der letzte Schritt ist der Aufruf dieses Compilers, zum Erstellen des endgültigen Bytecodes.

3.3.3. Zwischencode

Zur Speicherung aller relevanten Informationen über das Quellprogramm wird eine globale Datenstruktur aufgebaut. Dieses Objekt der Klasse **CompilerGlobals** wird von allen Visitoren benutzt, um Daten zwischenzuspeichern. Gleichzeitig dient sie als Schicht zwischen den Datenklassen und den Visitoren bzw. der Codegenerierung (siehe Abbildung 3.3). Als Vorlage für diese zentrale Klasse dient das Adapter-Design-Pattern[3]. Dieses sieht vor, dass der Adapter Methoden der adaptierten Klassen als zentrale Schnittstelle bereitstellt. Implementieren lässt es sich einfach, indem die Interfaces aus den Datenklassen extrahiert und diese dann in der **CompilerGlobals**-Klasse implementiert (IDEs wie Eclipse erzeugen die benötigten Methodensignaturen dann automatisch). Die Methodenaufrufe werden an die einzelnen Klassen weitergereicht. Die Klasse **CompilerGlobals** erfüllt somit die Funktion einer abstrakten Schnittstelle.

3.4. Gemeinsamkeiten zu NQC

Die entwickelte Programmiersprache soll eine größtmögliche Nähe zu NQC aufweisen, damit vorhandenene Dokumentation weiter genutzt werden kann, beziehungsweise nur in Teilen verändert werden muss. Dies ermöglicht den leichteren Umstieg für AnwenderInnen, die NQC schon benutzt haben. Die in NQC vorhandenen Sprachelemente sollen also soweit als möglich auch in NNQC vorhanden sein.

3.5. Unterschiede zu NQC

Die Unterschiede zu NQC sind hauptsächlich bedingt durch die Unterschiede in der Firmware zwischen den Mindstorms RCX und NXT Kits. Eine neue Version von NQC speziell für den NXT mag zukünftig anders aussehen. Tabelle 3.1 stellt die Unterschiede kurz dar (sie bezieht sich dabei auf Version 3.1.4 von NQC).

[3] Vergleiche [JAVAPATTERNS (1998)] , Seite 177.

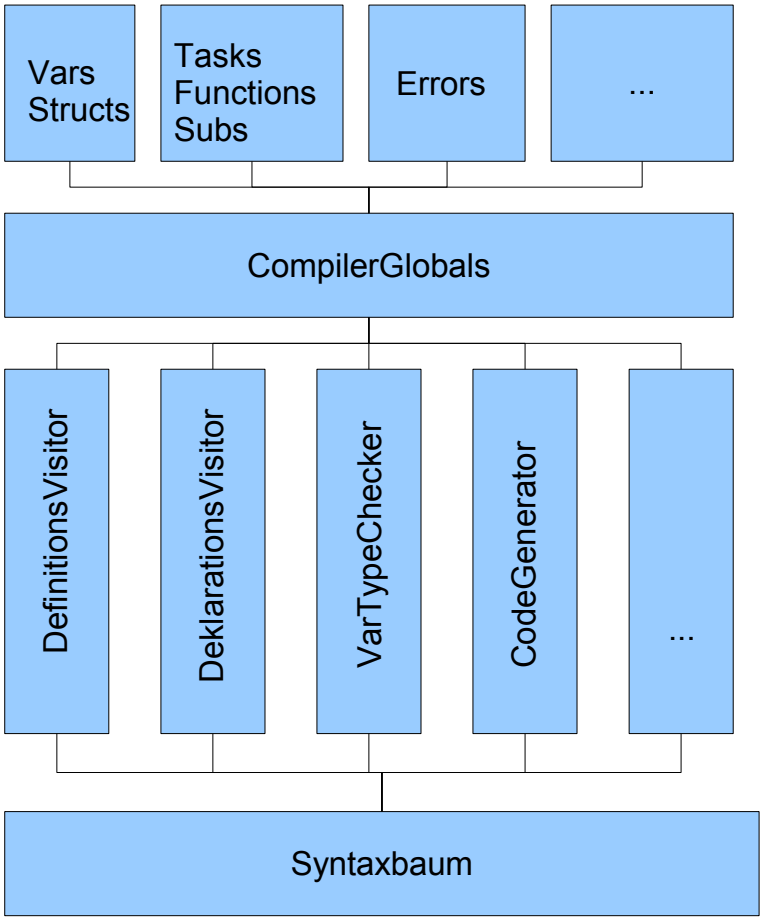

Abbildung 3.4.: Visitor(en) und Globale Daten

3.6. Einschränkungen von NNQC

Die folgenden Einschränkungen sind bedingt durch die NXT-Firmware.

- Die gesamte Anzahl an (auch lokalen und intern verwendeten) Variablen ist insgesamt auf 255 beschränkt.
- Es gibt keine Rückgabewerte von Funktionen. Durch Benutzung eines zusätzlichen Out-Parameters lässt sich dies aber ausgleichen. Wird bei der Definition der Funktion der Parameter mit & gekennzeichnet, so wird der Wert der übergebenen Variable verändert. Ansonsten wird er kopiert.
- Subroutinen und Tasks haben prinzipiell keine Übergabeparameter oder Rückgabewerte.

Tabelle 3.1.: Unterschied zwischen NQC und NNQC

Sprachelement	**NNQC**	**NQC**
Bitoperationen	nicht unterstützt	unterstützt
Strings	unterstützt	nicht unterstützt
Strukturen	unterstützt	nicht unterstützt
Lokale Variablen	unterstützt	nicht unterstützt
Anzahl Variablen	255	32[a]
Definition von globalen Variablen	innerhalb[b]	nur am Programmanfang

[a] Je nach Hardware Version
[b] Dies bedeutet innerhalb der gesamten Datei außerhalb von Sub-, Task- oder Funktions-Definitionen

- Die Benutzung von Rekursionen in Funktionen ist nicht möglich[4].
- Es darf derzeit nur einen Task geben, mit dem Namen "**main**". Scheduling von anderen Tasks ist nicht möglich[5], stattdessen sollten Subroutinen (Schlüsselwort **sub**) für eine Aufteilung benutzt werden.
- Schleifen (mit Ausnahme von *switch-case*) können nicht abgebrochen werden. Sie werden solange ausgeführt, bis der Vergleich fehlschlägt.

3.7. Ein Beispiel

Der folgende Code dient dazu, einen Eindruck der Syntax von NNQC zu vermitteln. Das folgende Programm zeigt zunächst den String "Hallo Welt"für eine Sekunde auf dem Display an.Danach werden zwei geschachtelte Schleifen durchlaufen, und die Werte der Schleifenvariablen angezeigt. Benutzt wird hierzu eine Bibliotheksfunktion *print*, die in der NNQC-Bibliothek vorhanden ist.

```
#include "print.h"

int a,b;
string ausgabe, tmp;

task main() {
    a = 0;
```

[4] Dies ist bedingt durch die Art der Funktionen, als Inline-Funktionen werden sie an der Stelle des Aufrufs in den Code eingefügt. Rekursion würde eine Endlosschleife bedeuten, bei der immer wieder erneut der Code eingefügt wird.
[5] NBC unterstützt nur Scheduling beim Beenden eines Threads/Tasks in der Form, das weiter Threads/Tasks gestartet werden können, nicht jedoch das Starten innerhalb eines laufenden Threads/Tasks. Das genaue Verhalten ist aufgrund der unvollständigen Dokumentation unklar.

```
    print("hallo welt");
    wait(1000);

    while ( a < 20) {
        b = 0;
        repeat(10) {
            ausgabe = "b: ";
            numtostr(tmp, b);
            strcat(ausgabe, tmp);
            numtostr(tmp, a);
            strcat(ausgabe, ",a: ", tmp);
            print(ausgabe);
            b++;
        }
        a++;
    }
}
```

3.8. Grammatik

Die komplette Syntax der Sprache NNQC wird mit Hilfe einer Grammatik dargestellt. Diese wird verwendet um mit Hilfe der Kombination aus JavaCC und JTB, Scanner, Parser und Syntaxbaum zu generieren.

3.8.1. Notation

Die angegebene Grammatik ist in EBNF dargestellt. Zusätzlich zur normalen EBNF Notation wird für **identifier**, **string_literal** und **digit** allerdings eine an reguläre Ausdrücke angelehnte Notation benutzt, um die komplette Aufzählung der enthaltenen Zeichen zu vermeiden[6]. Diese Erweiterung besteht im Wesentlichen aus der zusätzlichen Möglichkeit, Bereiche von Zeichen und das Nichtvorhandensein bestimmter Zeichenklassen zu definieren. Die folgende Übersicht stellt die vorhandenen Metazeichen dar:

- **[ ]** wird benutzt, um eine Menge von Zeichen zu definieren, die einzelnen Elemente werden innerhalb der Klammern durch ein Komma getrennt.

[6] in der formalen BNF besteht nur die Möglichkeit dies durch Aufzählen aller Elemente einer Menge zu tun.

- **“a” – “z"** bezeichnet einen Bereich in der Form **von – bis**, in diesem Falle also alle Zeichen von a bis z (a,b,c,d,. . . ,z).

- ()* zeigt eine optionale Wiederholung des Blockes an, der durch Klammern eingeschlossen ist, also beispielsweise ("wort")* für eine beliebige Anzahl des Strings “wort".

- ()+ zeigt eine Wiederholung an, bei der der Block mindestens einmal vorhanden sein muss.

- ˜ wird benutzt, um das folgende Zeichen oder die folgende Menge zu negieren, trifft also auf alle Zeichen zu, die in der folgende Menge **nicht** enthalten sind. Ein Beispiel ist der Ausdruck “[]", da die negierte Menge leer ist, trifft er auf alle Zeichen zu.

Diese Form der Regulären Ausdrücke ist kompatibel mit der von JavaCC akzeptierten Darstellung.

3.8.2. EBNF

Die folgende Grammatik beginnt zuerst mit der Deklaration einiger Terminals. Diese ließen sich zwar direkt in die Produktionen integrieren (abgesehen von den durch “quasi-reguläre” Ausdrück dargestellten), doch ist diese Form der Darstellung von der für JavaCC/JTB gebräuchlichen abgeleitet (siehe A.2). Der Grund für diese Deklarationen ist zum Einen die höhere Effizienz des generierten Parsers bei Verwendung einfacher Stringliterale, zum Anderen die Unterscheidbarkeit der erkannten Tokens anhand ihres Typs. Die gleichen Strings erhalten dabei vom Parser den gleichen (**int**) Typ zugewiesen. Der Vergleich zweier Integer Zahlen ist sehr viel schneller als der Vergleich zweier Strings.

```
kw_task = "task";
kw_function = "void" ;
kw_sub = "sub" ;
kw_exit = "exit" ;
kw_else = "else";
kw_if = "if";
kw_while = "while";
kw_do = "do" ;
kw_repeat = "repeat" ;
kw_switch = "switch" ;
kw_case = "case" ;
kw_default = "default" ;
kw_syscall = "syscall" ;
kw_struct = "struct" ;
kw_gettick = ``gettick'' ;
kw_int = "int" ;
kw_slong = "slong";
```

```
kw_ulong = "ulong";
kw_uword = "uword";
kw_sword = "sword";
kw_ubyte = "ubyte";
kw_sbyte = "sbyte";
kw_break = "break";
kw_return = "return"
kw_continue := "continue"
kw_default := "default"
kw_abs := "abs"
kw_sign := "sign"


op_eq = "==" ;
op_gt = ">" ;
op_lt = "<" ;
op_gteq = ">=" ;
op_lteq = "<=" ;
op_incr = "++" ;
op_decr = "\-\-" ;
as_norm = "=" ;
as_add = "+=" ;
as_sub = "\-=" ;
as_mul = "*=" ;
as_div = "/=";

COMMENT= "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")'' | "/*" (~["*"])* "*" ("*" | (~["*","/"]
          (~["*"])* "*"))* "/" ;
CHARACTER_LITERAL =  "\'" (~["\'","\\","\n","\r"] | "\\" (["n","t","b","r","f","\\","\'","
          \""] | ["0"\-"7"] (["0"\-"7"])? | ["0"\-"3"] ["0"\-"7"] ["0"\-"7"])) "\'" ;
STRING_LITERAL = "\"" ( ~["\"","\\","\n","\r"] | "\\" ( ["n","t","b","r","f","\\","\'","\""
          ] | ["0"\-"7"] (["0"\-"7"])?  | ["0"\-"3"] ["0"\-"7"] ["0"\-"7"] | ( ["\n","\r"]
           | "\r\n")))* "\"" ;
IDENTIFIER = (["a"\-"z","A"\-"Z"])(["a"\-"z","A"\-"Z","0"\-"9"])*>
DIGIT = (["0" \- "9"])(["0" \- "9"])* ;
HEXDIGIT = "0x" DIGIT ;

INPUT = ( VARIABLE_DECLARATION | STRUCT_DECLARATION_OR_DEFININITON | TASK_DECLARATION |
          FUNCTION_DECLARATION | SUB_DECLARATION ) {( VARIABLE_DECLARATION |
          STRUCT_DECLARATION_OR_DEFININITON | TASK_DECLARATION | FUNCTION_DECLARATION |
          SUB_DECLARATION )} <EOF>;
VARIABLE_DECLARATION = VARTYPE VARNAME  { "," VARNAME } ";" ;
VARNAME = identifier ;
LOCAL_VARIABLE_DECLARATIONS = LOCAL_VARIABLE_DECLARATION { LOCAL_VARIABLE_DECLARATION } ;
LOCAL_VARIABLE_DECLARATION = VARIABLE_DECLARATION | STRUCT_DECLARATION ``;";
TASK_DECLARATION␣=␣kw_task␣identifier␣"()"␣"{"␣[␣LOCAL_VARIABLE_DECLARATION␣]␣STATEMENTS␣"}
          "␣;
FUNCTION_DECLARATION␣=␣kw_function␣␣identifier␣"("␣[␣FUNCTION_ARGLIST␣]␣")"␣"{"␣[␣
          LOCAL_VARIABLE_DECLARATION␣]␣STATEMENTS␣"}"␣;
SUB_DECLARATION␣=␣kw_sub␣␣identifier␣"()"␣"{"␣[LOCAL_VARIABLE_DECLARATION]␣STATEMENTS␣"}"␣;
FUNCTION_ARGLIST␣=␣FUNCTION_ARG␣{␣","␣FUNCTION_ARG␣}␣;
FUNCTION_ARG␣=␣(VARIABLE_TYPE␣|␣kw_struct␣STRUCT_NAME)␣VARNAME␣[␣"&"␣]␣;
STRUCT_NAME␣=␣identifier␣;
STATEMENTS␣=␣STATEMENT␣{␣STATEMENT␣}␣;
STATEMENT␣=␣ASSIGNMENT␣|␣IFLOOP␣|␣DOLOOP␣|␣REPEATLOOP␣|␣SWITCHLOOP␣|␣WHILELOOP␣|␣SYSCALL␣|␣
          EXIT␣|␣NUMTOSTR␣|␣STRCAT␣|␣GET␣|␣SET␣|␣GETTICK␣|␣SUBFUNCTIONCALL␣|␣VAR_INC_DECR␣
          ;
EXIT␣=␣kw_exit␣"()"␣";"␣;
GETTICK␣=␣kw_gettick␣``(" STRUCTPART_OR_VARNAME ``)'' ``;'' ;
STRUCTPART_OR_VARNAME = STRUCTPART | VARNAME ;
NUMTOSTR = kw_numtostr "(" STRUCTPART_OR_VARNAME "," (DIGIT|HEXDIGIT|STRUCTPART_OR_VARNAME)
           ")" ";" ;
```

```
STRCAT = kw_strcat "(" STRUCTPART_OR_VARNAME "," (STRING_LITERAL|STRUCTPART_OR_VARNAME) { "
          "," (STRING_LITERAL|STRUCTPART_OR_VARNAME) } ")" ";" ;
SYSCALL = kw_syscall "(" (DIGIT|HEXDIGIT) "," STRUCTPART_OR_VARNAME ")" ";" ;
GET = kw_get "(" STRUCTPART_OR_VARNAME "," (DIGIT|HEXDIGIT) ")" ";" ;
SET = kw_set "(" (DIGIT|HEXDIGIT)"," ARITH_EXPRESSION ")" ";" ;
SUBFUNCTIONCALL = identifier "(" [ ( ARITH_EXPRESSION| STRING_LITERAL ) { "," (
          ARITH_EXPRESSION| STRING_LITERAL ) } ] ")" ";" ;
IFLOOP = kw_if "(" COMPARE_EXPRESSION ")" "{" STATEMENTS "}" [ kw_else "{" STATEMENTS "}" ]
          ;
WHILELOOP = kw_while "(" COMPARE_EXPRESSION ")" "{" STATEMENTS "}" ;
REPEATLOOP = kw_repeat "(" ARITH_EXPRESSION ")" "{" STATEMENTS "}" ;
SWITCHLOOP = kw_switch "(" ARITH_EXPRESSION ")" "{" { kw_case DIGIT":" (STATEMENTS|kw_break
           ";") } [ kw_default ":" (STATEMENTS|kw_break ";") ] "}" ;
DOLOOP = kw_do "{" STATEMENTS "}" kw_while "(" COMPARE_EXPRESSION ")"
VARIABLE_TYPE = kw_int|kw_string|kw_dword|kw_sword|kw_sbyte|kw_dbyte|kw_byte
COMPARE_EXPRESSION = ARITH_EXPRESSION (op_eq|op_gt|op_lt|op_gteq|op_lteq|op_neq)
          ARITH_EXPRESSION [ (OP_AND|OP_OR) COMPARE_EXPRESSION ]
VAR_INCR_DECR = STRUCTPART_OR_VARNAME (op_incr|op_decr)''';'' ;
ASSIGNMENT = ( STRUCTPART_OR_VARNAME ) (as_norm|as_add|as_sub|as_mul|as_div)
          ARITH_EXPRESSION|string_literal ";" ;
STRUCTPART = VARIABLE "." identifier { "." identifier } ;
ARITH_FUNCTION = op_abs|op_sign ;
ARITH_EXPRESSION = ARITH_TERM { (op_add|op_sub) ARITH_TERM } ;
ARITH_SIGN = op_add|op_sub ;
ARITH_TERM = ARITH_FACTOR { (op_mul|op_div|op_mod) ARITH_FACTOR } ;
ARITH_FACTOR = [ ARITH_FUNCTION ] "(" ARITH_EXPRESSION ")" | [ ARITH_SIGN ] (
          STRUCTPART_OR_VARNAME [op_incr|op_decr] | DIGIT|HEXDIGIT)
STRUCT_DECLARATION_OR_DEFINITION = STRUCT_DEFINITION | STRUCT_DECLARATION ";" ;
STRUCT_DEFINITION = kw_struct identifier "{" (VARIABLE_TYPE | kw_struct identifier )
          identifier ";" { (VARIABLE_TYPE | kw_struct identifier ) identifier ";" } "}" ;
STRUCT_DECLARATION = kw_struct identifier VARIABLE { "," VARIABLE }
```

3.8.3. Nachweis der LL(k) Eigenschaften

Da JavaCC eine LL(k) Grammatik voraussetzt, muss sichergestellt sein, dass die Grammatik auch den Anforderungen genügt.

Überführung in BNF

Zum formalen Nachweis der LL(k) Eigenschaften der Grammatik ist eine Überführung in BNF erforderlich. Dazu sind die iterativen Teile der EBNF Grammatik (also die Blöcke, die in geschweiften Klammern stehen) in eine Rekursion zu überführen. Beispielhaft wird dies anhand **SUBFUNCTIONCALL** hier dargestellt:

```
SUBFUNCTIONCALL = identifier "(" [ ( ARITH_EXPRESSION| STRING_LITERAL ) { "," (
          ARITH_EXPRESSION| STRING_LITERAL ) } ] ")" ";" ;
```

wird durch Umwandlung der iterativen Komponenten zu:

```
SUBFUNCTIONCALL = identifier "(" SUBFUNCTIONCALL_O ")" ";" ;
SUBFUNCTIONCALL_O = ( ARITH_EXPRESSION| STRING_LITERAL ) SUBFUNCTIONCALL_O_R
SUBFUNCTIONCALL_O_R = "," ( ARITH_EXPRESSION| STRING_LITERAL ) SUBFUNTIONCALL_O_R | \
          epsilon
```

Die Konvertierung der gesamten EBNF-Grammatik findet sich im Anhang (A.1). Dabei ist anzumerken, dass die dort verwendete BNF zusätzlich die schon vorgestellte Form der regulären Ausdrücke (siehe 3.8.1) verwendet. Für die Entscheidung, ob die Grammatik LL(k) ist, spielt die Darstellung der Terminals keine Rolle.

Eigenschaften von LL(K) Grammatiken

Aus einer BNF Grammatik lässt sich mit Hilfe der Funktionen FIRST und FOLLOW eine Parse-Tabelle ermitteln (siehe [DRACHENBUCH (1999)], S.231). Enthält diese keine mehrfachen Einträge an der gleichen Stelle, so ist die Grammatik vom Typ LL(1). Sind mehrfache Einträge vorhanden, so ist zu prüfen, ob die Grammatik durch einen größeren LOOKAHEAD (k > 1) die Mehrdeutigkeit verliert, beziehungsweise ob sich die Grammatik durch Umformung (Linksfaktorisierung) in eine LL(1) Grammatik überführen lässt.

Parsetabelle

Aufgrund der (visuellen und inhaltlichen) Komplexität der Parsetabelle wird auf eine komplette Darstellung verzichtet. Die Überprüfung, ob die Grammatik LL(1) Eigenschaften besitzt, lässt sich auch mit Software durchführen.
Das ohnehin verwendete JavaCC führt eine Überprüfung der FIRST- bzw. FOLLOW-Mengen automatisch beim Parsevorgang durch, und warnt Anwender gegebenfalls, an welchen Produktionen mit einem LOOKAHEAD von eins nicht entschieden werden kann, welche Produktion zu wählen ist. Die Parsetabelle (Tabelle 3.2) wird für diese Fälle als Ausschnitt[7] angegeben. Ein Durchlauf mit der Grammatik aus A.2, ohne die dort vorhandenen *LOOKAHEAD* Anweisungen, fördert folgende Problemstellen zu Tage:

Tabelle 3.2.: Auszug aus der Parsetabelle

Non-Terminal	**Terminal**							
	identifier	kw_if	kw_do	...	kw_exit	kw_numtostr	kw_strcat	kw_gettick
STATEMENT	ASSIGNMENT VAR_INC_DECR SUBFUNCTIONCALL	IFLOOP	DOLOOP	...	EXIT	NUMTOSTR	STRCAT	GETTICK
STRUCTPART_OR_VARNAME	STRUCTPART VARNAME							
STRUCT_DECLARATION_OR_DEFINITON	STRUCT_DECLARATION STRUCT_DEFINITION							

[7] Dabei sind nicht alle Produktionen für alle Terminals angegeben, die Terminals *kw_switch*, *kw_while*, *kw_repeat*, *kw_do* und *kw_numtostr* fehlen.

- **STATEMENT:** Es verwundert nicht weiter, dass es an dieser Stelle zu Problemen kommt, *STATEMENT* ist eine der zentralen Produktionen dieser Grammatik. Ganz konkret ist die Entscheidung zwischen *ASSIGNMENT*, *VAR_INC_DECR* und *SUBFUNCTIONCALL* mit einem Lookahead von eins nicht zu treffen. Auch eine Erhöhung des Lookahead Wertes auf zwei oder drei schafft nur in einigen Fällen Abhilfe. Der Aufruf einer Subroutine oder Funktion lässt sich mit einem Lookahead von zwei zuverlässig unterscheiden, und zwar an der folgenden öffnenden Klammer.
  Die Unterscheidung zwischen Inkrement/Dekrement (*VAR_INC_DECR*) und der Zuweisung (*ASSIGNMENT*) ist mit einem festen Lookahead-Wert theoretisch nicht immer, praktisch sehr wohl zu treffen. Der Grund für das Problem ist, dass beide mit einer Struktur beginnen können, die (theoretisch) beliebig oft ein Konstrukt der Form *identifier* "." *identifier* enthalten kann. Die Entscheidung, ob es sich um eine Zuweisung (also als nächstes ein Zuweisungs-Operator folgt) oder ein Inkrement/Dekrement (also als nächstes ein **++** oder **- -** folgt) ist erst nach der Abarbeitung des Strukturzugriffs zu treffen. Theoretisch kann der nötige Lookahead-Wert also unendlich groß sein, in der Praxis kommen Strukturen mit einer unendlichen Schachtelungstiefe jedoch nicht vor. Da der Lookahead-Wert nicht fest bestimmt werden kann, bietet es sich an, die Entscheidung anhand eines variablen, semantischen Lookaheads vorzunehmen. Bei diesem wird JavaCC angewiesen, solange den Lookahead fortzusetzten, bis die Produktion identifiziert werden kann (also klar ist, ob ein Zuweisungs- oder ein In-/Dekrement Operator folgt). Es wird dafür kein fester Wert für k gesetzt.
- **STRUCTPART_OR_VARNAME:** Die First-Mengen von STRUCTPART und *VARNAME*, enthalten beide den Token identifier. Ein Lookahead von eins ist hier nicht ausreichend, einer von zwei jedoch schon, da sich dann der *STRUCTPART* durch den folgenden Punkt zum Zugriff innerhalb der Struktur von *VARNAME* (das immer nur aus einem *identifier* besteht) unterscheiden. Durch Linksfaktorisierung von **identifier** ist das Problem ebenfalls zu lösen. Darunter würde aber Verständlichkeit der Grammatik leiden.
- **STRUCT_DECLARATION_OR_DEFINITON:** Die beiden Produktionen STRUCT_DECLARATION und STRUCT_DEFINITION haben für die nächsten zwei Tokens identische FIRST-Mengen. Beide beginnen mit *struct identifier*, erst danach lassen sie sich unterscheiden. Dieses Problem ließe sich durch eine Linksfaktorisierung auflösen, jedoch ist die Grammatik so lesbarer. Dies ist bei der Erstellung der Visitoren vorteilhaft. Ein lokal auf 3 erhöhter Lookahead-Wert erfüllt den gleichen Zweck.

An zwei der drei Stellen mit Problemenen in der Grammatik läßt sich eine Eindeutigkeit mit einem Lookahead-Wert von eins also leicht herstellen. Im Falle von *STATEMENT* beziehungsweise gelingt dies jedoch nicht, ohne die Grammatik schwer lesbar zu machen.

3.8.4. Kontextfreie Grammatiken und Programmiersprachen

Programmiersprachen, die eine Deklaration der Variablen vor der Benutzung erfordern, sind nicht kontextfrei[8]. Da im Falle von NNQC die Variable deklariert werden muss, dies aber in der Grammatik nicht deutlich gemacht werden kann (in beiden Fällen wird dort lediglich “Identifier"benutzt), entspricht die Grammatik nicht der Programmiersprache. Eine solche Sprache lässt sich aber durch eine kontextfreie Grammatik und weitere semantische Überprüfungen darstellen.

3.8.5. Typische Probleme

Normalerweise kommt es bei C-ähnlichen Sprachen wie NNQC zu Problemen bei der Unterscheidung von Zuweisungen von arithmetischen bzw. logischen Ausdrücken, also etwa ein *boolean b; b = 2 * 3 +4 < 12;*. Dabei ist die Entscheidung, ob es sich beim Term auf der rechten Seite um einen arithmetischen bzw. einen logischen Ausdruck handelt mit einem LOOKAHEAD von 1 nicht zu treffen. Auch sonst macht ein fester LOOKAHEAD Probleme.
Eine Möglichkeit wäre die Überprüfung des Typs der Variablen auf der linken Seite, **boolean** hieße in diesem Fall, dass auf der rechten Seite ein logischer Ausdruck stehen muss. NNQC verfügt allerdings über keinen Datentyp boolean, wie bei anderen C-ähnlichen Sprachen ist das Äquivalent zu **boolean** der Typ **int** (oder **byte**, **word**), mit der Verabredung: 0 entspricht **false** und alles größer als 0 ist **true**. Dadurch entfällt das Problem von selbst, da die Zuweisung in jedem Fall semantisch korrekt ist, sofern es sich beim Typ der Variable auf der linken Seite der Zuweisung um eine numerischen Datentyp handelt.
Das verbleibende Problem ist die Erkennung einer arithmetischen bzw. booleschen Zuweisung auf eine Variable vom Typ *String* (Der Scanner/Parser sieht nur identifier, und weiß nichts vom Typ einer Variablen). Dies muss über die Semantikprüfung mit Hilfe eines Visitors abgefangen werden.

3.8.6. Dangling-Else-Problem

Das “Dangling-Else-Problem” ist ein in der Entwicklung von Programmiersprachen oft auftauchendes Problem, sobald bedingte Anweisungen wie *if* durch optionale Zweige (*else*) ergänzt werden können. Die Frage ist schlicht, zu welchem *if* gehört das *else*? In diesem Punkt verlässt NNQC die Nähe zur C-Syntax, und fordert das explizite Setzen von geschweiften Klammern um *else* und *if* Blöcke. Dadurch ist eindeutig erkennbar, zu welchem *if* das *else* gehört.

[8] Vergleiche [DRACHENBUCH (1999)] , Seite 218.

3.9. Präprozessor

Um dem eigentlichen Compiler Arbeit abzunehmen und dem Anwender die Arbeit zu erleichtern, ist ein Präprozessor sinnvoll. Dieser hat die Funktion, den eventuell in mehrere Dateien aufgeteilten Programmcode zu einem Ganzen zusammenzufügen und Textersetzungen (von Konstanten oder Makros) vorzunehmen.

3.9.1. Funktionsumfang

Die gewünschten Funktionen des Präprozessors sind: *import* und *define* (für Konstanten und Makrofunktionen), auf Anweisungen wie *ifdef* und *undef* wird verzichtet, da sie nicht unbedingt benötigt werden.

3.9.2. Verwendete Werkzeuge

Der Präprozessor soll, wie der Compiler auch, mit Hilfe von JavaCC und JTB entstehen. Dies hat den Vorteil, dass keine neue Technik eingesetzt werden muss. Es wäre prinzipiell auch möglich, bereits existierende Präprozessoren zu verwenden, da als Schnittstelle lediglich die Ausgabe des kompletten Programmtextes als String (oder über eine temporäre Datei) benötigt wird. Das Entwickeln eines eigenen Präprozessors bietet jedoch den Vorteil, die Positionen von Konstrukten im ursprünglichen Text (d.h. an welcher Stelle in welcher Datei) verfügbar zu halten, um präzisere Fehlermeldungen zu ermöglichen. Hinzu kommt, dass sich der Aufwand, einen eigenen Präprozessor zu entwickeln, sehr in Grenzen halten sollte.

3.9.3. Grammatik des Präprozessors

Diese Grammatik ist, bis auf die Definition der Tokens in Form eines regulären Ausdrucks, in EBNF dargestellt. Die Definition der verwendeten Darstellung für die regulären Ausdrücke findet sich in Abschnitt 3.8.1.

```
ZEILENENDE = ``\n'' | ``\r'' | ``\r\n'' ;
single_line_comment = ``// [^ZEILENENDE]'' ;
multiline_comment = ``/*'' [^"*"] ``*/'' ;
pre_begin = ``#'' ;
include = ``include'' ;
define = ``define'' ;
filename = ([ "a" - "z", "0" - "9" , "_", "-", "A"- "Z", "/", ".", "\\"])+ ;
digit = (["0" - "9"]) + ;
identifier = (["a"-"z","A"-"Z","_","-"])(["a"-"z","A"-"Z","_","-"]|<DIGIT>)* ;
functionbody = ~["\n","\r"]
rest = ~["#"] ;
```

```
INPUT = PRE_STATEMENT | CODEBLOCK | COMMENT ;
COMMENT = single_line_comment | multi_line_comment ;
PRE_STATEMENT = pre_begin (INCLUDE | DEFINE_FUNTION | DEFINE_LITERAL);
INCLUDE = include ``'''' filename ``'''' ;
DEFINE_LITERAL = define identifier (digit|string_literal|identifier) ;
DEFINE_FUNCTION = define identifier ``("␣[identifier␣{","␣identifier}]␣``)''␣functionbody␣;
CODEBLOCK␣=␣rest␣{rest}␣;
```

Diese Grammatik ist ebenfalls nicht vom Typ LL(1). Dies wird beim PRE_STATEMENT deutlich. Die folgenden DEFINE_LITERAL und DEFINE_FUNCTION haben die gleichen zwei Tokens am Anfang. Durch einen (lokal) erhöhten Lookahead-Wert von drei lässt sich das Problem allerdings beheben. Wie auch schon bei der Compiler-Grammatik, ist der Grund für diese Verletzung der LL(1)-Eigenschaft die bessere Lesbarkeit. Durch (Links-) Faktorisieren läßt sich das Problem beseitigen, ohne den Lookahead erhöhen zu müssen:

```
DEFINE_LITERAL_OR_FUNCTION = DEFINE INDENTIFIER
DEFINE_LITERAL = DEFINE IDENTIFIER (DIGIT|STRING_LITERAL|IDENTIFIER)
DEFINE_FUNCTION = DEFINE IDENTIFIER ``("␣[IDENTIFIER␣{","␣IDENTIFIER}]␣``)''␣FUNCTIONBODY␣;
```

Die für JacaCC/JTB verwendete Grammatik befindet sich im Anhang (A.3). Sie weist gegenüber der einfachen, hier angegebenen EBNF-Grammatik den Unterschied auf, das verschiedene Lexikalische Stati für unterschiedliche Zustände des Scanners verwendet werden. Ansonsten würde es zu Problemen mit den regulären Ausdrücken zur Stringerkennung kommen, da ein Zeichen nicht mehrfach vorkommen kann[9].

3.9.4. Fehlerbehandlung

Die Fehlerbehandlung im Präprozessor erfolgt, analog zu der im eigentlichen Compiler, über die Fehlerliste in der globalen Datenstruktur **CompilerGlobals**, d.h. nicht mittels Exceptions (Vergleiche Abschnitt 3.12). Die vom Parser generierten Parse-Exceptions sind aus diesem Grunde abzufangen und in Fehler in der zentralen Fehlerliste umzuwandeln. Im Folgenden werden typische Fehler und deren Erkennung erläutert.

3.9.5. Include-Zyklen

Durch die *include* Anweisung des Präprozessors besteht die Gefahr, dass sich (Endlos-) Schleifen von Dateien ergeben, die sich gegenseitig einbinden. Eine Lösung zur Erkennung dieses Problems ist die Benutzung eines Baumes, der die Include-Hierarchie abbildet. Eine per *include* hinzugefügte Datei wird als Kind an die aktuelle Datei angehängt. Für jeden hinzugefügten Knoten wird im Pfad zu Wurzel geprüft, ob sich der aktuelle Knoten

[9] Beispielsweise würde ein eigenes Token Komma=","verhindern, das ein Komma im Ausdruck REST= [] (also alle Zeichen) vorkommen kann. Da diese Tokens spezifisch für einen Zustand des Scanners sind, teilt man die sich überschneidenden Tokens in unterschiedliche Zustände.

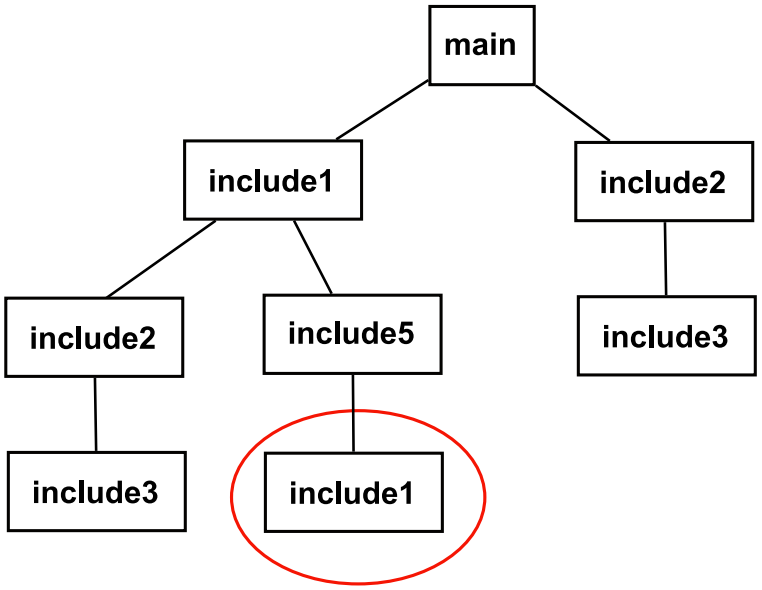

Abbildung 3.5.: Zyklenerkennung beim include von Dateien

schon einmal im Pfad befindet. Ist dies der Fall, so haben wir einen Zyklus. In Abbildung 3.5 ist dies in einem Beispiel dargestellt. Bei der Pfadüberprüfung für den rot markierten (*include1*) Knoten wird der Zyklus festgestellt. Die beiden doppelt vorhandenen Teilbäume mit *include2* und *include3* stellen hingegen keinen Zyklus da, sie tauchen auf dem Weg zur Wurzel (jeweils) nur einmal auf.

3.9.6. Zeilennummernproblematik

Durch den Präprozessor werden eventuell die Zeilennummern im Quelltext geändert, im vorliegenden Fall bei *include*-Direktiven[10]. Durch das Zusammenfügen der einzelnen Dateien zu einer Datei, die den gesamten Programmtext enthält, verschieben sich die Zeilennummern. Dadurch wird es schwierig, bei einem Fehler die korrekte Zeilennummer auszugeben.
Eine korrekte Zeilennummer und ein Dateiname sind jedoch Vorraussetzung, damit der Programmierer den Fehler in annehmbarer Zeit finden kann. Zur Lösung dieses Problems gibt es mehrere Möglichkeiten:

- Integration des Präprozessors direkt in den Compiler, die einzelnen, durch *include* hinzukommenden Programmcodeblöcke werden direkt vor dem Zusammenfügen syn-

[10] *define*-Direktiven sind von der Präprozessorgrammatik auf einzeilige Anweisungen beschränkt, somit bleibt die Zeilenanzahl gleich.

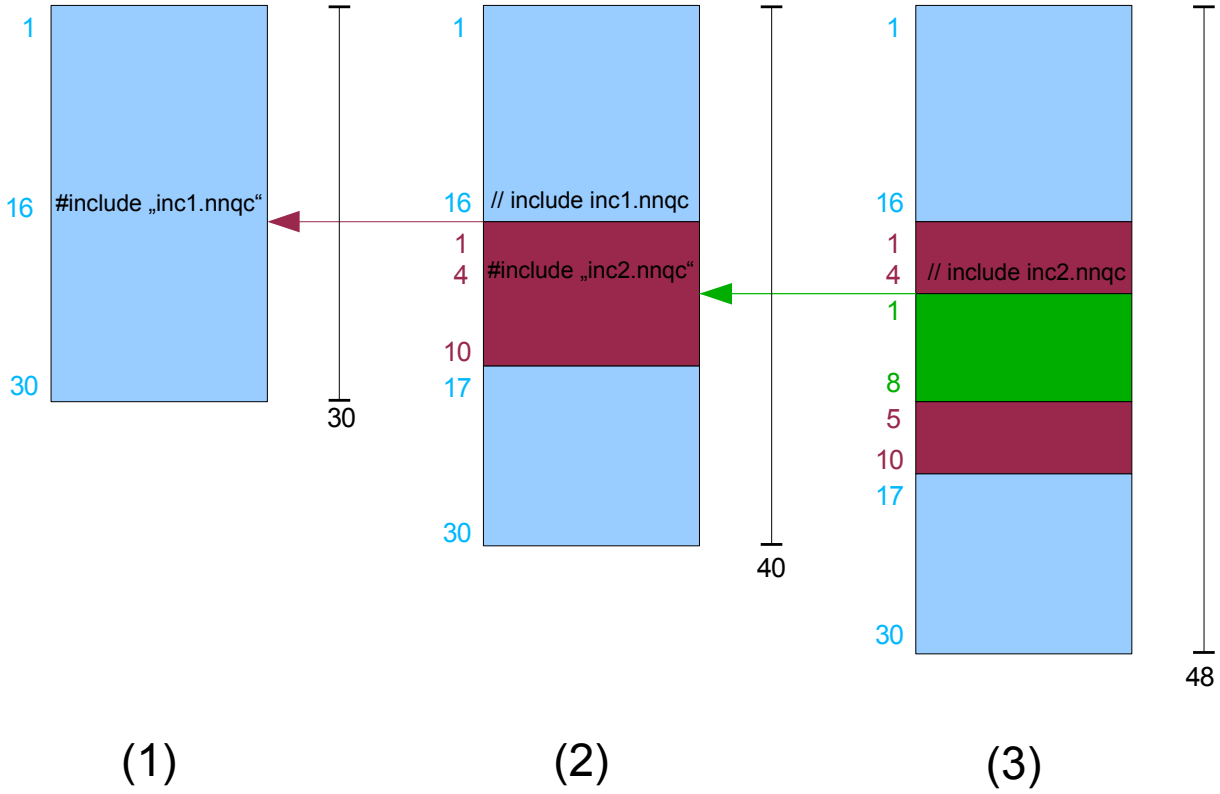

Abbildung 3.6.: Include von Dateien

taktisch analysiert, Fehler sind sofort zuortbar. Für bei diesem Vorgang nicht erkennbare, semantische Fehler, bleibt die Problematik allerdings bestehen.

- Einfügen von (internen) Markierungen in den Programmtext in Form von Kommentaren, die vom Parser (nicht vom Scanner, dies würde den späteren Zugriff verhindern) im Normalfall ignoriert werden. Dies setzt eine Änderung an der vorliegenden Grammatik von NNQC voraus, derzeit werden alle Kommentare im Scanner ausgefiltert.
- Das Mitprotokollieren in einer entsprechenden Datenstruktur, welche die Zuordnung von Zeilennummern im endgültigen Programmtext zur Ursprungsdatei und Zeilennummer vornimmt.

Für diesen Compiler soll die letzte Variante (Mitprotokollieren) zum Einsatz kommen, da sie ohne große Änderungen am Compiler umgesetzt werden kann. Lediglich der Präprozessor selbst muss solch eine Datenstruktur pflegen, die Methoden zur Ausgabe von Fehlern müssen diese ebenso berücksichtigen.
Die dazu verwendete Datenstruktur besteht aus einer Liste von Codeblöcken. Ein solcher Block verfügt über die Information, aus welcher Datei er stammt, in welcher Zeilennummer er dort beginnt und welchen Bereich (Start- und Endpunkt) er im endgültigen Quelltext

abdeckt.
In Abbildung 3.9.6 ist dies beispielhaft dargestellt. Position 1 zeigt dabei den Ausgangszustand. Die Zahlen in den Farben der – als Blöcke dargestellten – Dateien zeigen dabei die Position innerhalb der Datei. Die Gesamtlänge (auf die sich Fehlermeldungen des späteren Compilerlaufs beziehen werden) ist jeweils rechts in schwarz angegeben.
Beim Einfügen der Datei *inc1.nnqc* (Position 2) wird zunächst die *include*- Direktive durch einen entsprechenden Kommentar ersetzt. Dies hat lediglich den Zweck, die Länge der Datei, die das *include* beinhaltet, beizubehalten, um die Verschiebung von Zeilennummern für nachfolgenden Code aus derselben Datei später bestimmen zu können. Danach wird die Länge des eingefügten Codeblocks bestimmt.
Gibt es Codeblöcke, die hinter dem Aktuellen liegen, so ist ihre Position um diese Länge zu verschieben (siehe Position 3, der zweite “blaue"Block wird verschoben).
Der aktuelle Codeblock, der das *include* enthielt, wird nun in 2 Codeblöcke aufgeteilt, der 2. Block entsprechend verschoben und der neue Block eingefügt. Im neuen 2. Block ist der neue Startpunkt innerhalb der originalen Quelldatei zu bestimmen (die Zeile nach dem *include*), der Startpunkt in der Gesamtquelldatei ist um die Länge des neuen Codeblocks zu verschieben.

Sind diese Operationen durchgeführt, so muss ein neuer Scan/Parse-Vorgang des Präprozessors auf dem jetzt enstandenen Gesamtquelltext ausgelöst werden, damit in dem eingefügten Code enthaltende “include” Direktiven beachtet werden. Dies wird solange wiederholt, bis keine includes mehr vorhanden sind.

3.10. Zwischencode

Als Bindeglied zwischen Frontend und Backend des Compilers dient ein Objekt vom Typ **CompilerGlobals**. Dieses stellt alle zur Erzeugung des Zielcodes benötigten Daten zur Verfügung. Auf eine Überführung in einen unabhängigen Zwischencode (wie etwa 3-Adress-Code oder einen eigenen Baum) wird verzichtet. Der Hauptgrund auf einen unabhängigen Zwischencode zu verzichten, liegt in der schnelleren Erreichung des Ziels, einen lauffähigen Compiler zu erzeugen. Zudem liegen die Daten schon weitgehend in einem Format vor, das die Erzeugung von NBC-Code erlaubt. Die zur Erzeugung der Grundstruktur des NBC-Zielcodes nötigen Informationen sind in den entsprechenden Datenstrukturen (Funktionen, Subroutinen, Tasks, Variablen, Strukturdefinitionen) abrufbar. Die verbleibenden Anweisungen zum Komplettieren des Zielcodes sind in der gleichen Form wie sie vom Parser im Syntaxbaum abgelegt wurden, vorhanden. Dabei ist jede Anweisung vom Typ **statement** ein Teilbaum des AST.
Ein weiterer Grund für den Verzicht auf eine unabhängige Zwischensprache ist, dass die

Ausrichtung dieses Compilers nicht primär auf der Unterstützung verschiedener Zielsysteme liegt und die Quellsprache keinen Anspruch auf Allgemeingültigkeit hat. Die einzelnen Anweisungen können fast alle problemlos in NBC-Code umgesetzt werden, lediglich für die arithmetischen Ausdrücke ist eine weitere Vorverarbeitung nötig (siehe 3.17.9).

3.11. Aufruf des NBC Compilers

Zum Aufruf des NBC-Compilers wird eine Klasse zur Kapselung des Aufrufs geschaffen. Sollte es während des NBC-Compilerlaufs zu Fehlern kommen, ist es aufwendig, diese Fehler auf die entsprechenden Zeilen im NNQC-Code zurückzuführen. Es ließe sich zwar eine ähnliche Technik wie beim Präprozessor anwenden, allerdings ist die Wahrscheinlichkeit groß, dass ein interner Fehler vorliegt. Ein solcher Fehler wird durch den vom Compiler erzeugten NBC-Code verursacht. Dies lässt sich dann nur mit einer Korrektur im Sourcecode des Compilers beheben. Eine Fehlermeldung, die in diese Richtung weist und möglichst viele Informationen mitliefert, ist daher sinnvoll.

3.12. Fehlerbehandlung

Die Fehlerbehandlung erfolgt primär durch eine, in der globalen Datenstruktur **CompilerGlobals**, verwaltete Liste an Fehlern und Warnungen. Durch die Problematik keine eigenen Exceptions innerhalb der *visit*-Methoden eines Visitors (und damit im Visitor selber) benutzen zu können, werden stattdessen Fehler (vom Typ **Error** oder **Warning**) generiert, und diese an die Fehlerliste übergeben. Durch Abfragen der Länge der Fehlerliste wird zwischen den einzelnen Schritten im Hauptprogramm entschieden, ob Fehler vorliegen. Sind Fehler vorhanden wird der nächste Schritt im Hauptprogramm nicht mehr ausgeführt, sondern das Programm mit Ausgabe der Fehlermeldungen (und aufgelaufenen Warnungen) beendet. Um dies einheitlich zu halten, sind die vom Scanner und Parser generierten Parse-Exceptions abzufangen und in einen internen Fehler umzuwandeln. Das Parsing sollte dann an der nächsten sinnvollen Stelle weitergeführt werden[11].

3.13. Bibliotheksfunktionen

NNQC verfügt nur über einen recht beschränkten Umfang, der allerdings die Erweiterung über das Entwickeln von Programmbibliotheken unterstützt. Erleichtert wird dies durch die

[11] Als Stichwort sei hier der sogenannte Panikmodus genannt, bei dem alle Symbole überprungen werden, bis ein Element aus einer Synchronisationsmenge gefunden wird, nach dem der Parse-Vorgang fortgesetzt werden kann (oft das nächste “;").

Unterstützung des zentralen NBC Befehls *syscall*. Fast alle Funktionalität, die die Ein- und Ausgabe betreffen, sind in NBC über den Systemaufruf realisiert. Das beinhaltet auch das Anlegen und Verändern von Dateien, den Aufbau einer Kommunikation über Bluetooth, etc. Durch die Unterstützung dieser Funktion in NNQC ist es möglich, die dafür generierten Strukturen an den NBC-Systemcall weiterzureichen. Die Einbindung der einzelnen Bibliotheksfunktionen erfolgt dann durch *include* Anweisungen an den Präprozessor.
Die Deklaration von Variablen und Strukturen kann für diese Bibliotheksfunktionen entweder global (also im Quellcode vor der eigentlichen Funktion) oder innerhalb – als lokale Variable – erfolgen. Beides ist nach der Grammatik zulässig und wird unterstützt. Eine Abtrennung der Variablen- beziehungsweise Strukurdeklaration in eine weitere include-Datei ist somit unnötig.
Werden keine lokalen Variablen benötigt, so ist eine Funktion in der Regel auch als Makro für den Präprozessor realisierbar.

3.14. Funktionen, Tasks und Subroutinen

Funktionen, Tasks und Subroutinen werden über CompilerGlobals durch eine Liste von Objekten dargestellt. Diese enthalten die folgenden Informationen:

- Den Namen der Funktion/Subroutine bzw. des Tasks.
- Eine Liste der lokalen Variablen.
- Im Falle einer Funktion die Parameter (in Form von Objekten des Typs **Variable**).
- Eine Liste der Anweisungen (Typ **statement**).

Zum Ermitteln des kompletten Namens einer Variablen ist die Methode *getRealname* zu implementieren, diese liefert den endgültigen, global gültigen Namen einer Variable. Dazu wird in der Liste der lokalen Variablen überprüft, ob dort ein Eintrag mit entsprechendem Namen existiert. Existiert er nicht, ist die Variable global (zur Namensgebung bei lokalen Variablen siehe 3.15).
Eine Funktion unterscheidet sich von Subroutinen und tasks hauptsächlich durch die Möglichkeit, Parameter zu übergeben. Zu den Unterschieden in der Codegenerierung siehe Abschnitt 3.17.3.
Der rekursive Aufruf von Subroutinen wird von NBC nicht unterstützt. Ein Algorithmus wie in 3.9.5 verhindert auch rekursive Aufrufe mit mehreren Zwischenstationen.

3.15. Lokale Variablen

Die Sprache NNQC unterstützt die Benutzung lokaler Variablen. Diese sind nur innerhalb eines Tasks, einer Funktion oder Subroutine gültig. Lokale Variablen verlieren nicht ihren Wert beim Verlassen einer Funktion oder Subroutine/Task, da sie über global definierte Variablen abgebildet werden (etwas anderes unterstützt NBC nicht). Das bedeutet, dass die Werte der Variablen zwischen den Aufrufen derselben Funktion erhalten bleiben. Die Namensgebung der hierfür verwendeten globalen Variablen verhindet den direkten Zugriff von NNQC aus. [12]. Eine weitere Möglichkeit diese Variablen abzubilden, wäre die Benutzung eines Stacks, beziehungsweise im Falle von NBC (das selber keine Stacks kennt) eines globalen Arrays für (beispielsweise) “signed Integer", und der Zugriff über den Index, der zur Kompilierzeit festgelegt werden kann. Dies würde die begrenzte Gesamtanzahl an zulässigen Variablen entlasten, allerdings wäre es für Strukturen nicht möglich, da für jede Struktur dann ein eigenes Array angelegt werden müsste (NBC unterstützt nur den Zugriff durch call-by-name. Zudem gibt es keine Unterstützung von Unions). Da die Anzahl von 255 Variablen deutlich über der Anzahl der von NQC unterstützen 32 liegt, wird vorerst darauf verzichtet, lokale Variablen durch ein Array abzubilden.

3.16. Ablaufsteuerung

Der gesamte Compilerlauf wird von einem Objekt der Klasse **Main** gesteuert. Dies ruft die einzelnen Teile des Compilers (Präprozessor, Parser, Visitoren, Codegenerierung) auf, die Abarbeitung erfolgt sequentiell. Eine Nebenläufigkeit von Teilen des Codes ist nicht zu erwarten, eventuelle Maßnahmen wie das Sicherstellen der Threadsicherheit durch Lockingmechanismen erfolgt deshalb nicht.

3.17. Codegenerierung

Die Generierung des NBC-Codes erfolgt nicht über einen Visitor, der den kompletten AST traversiert. Stattdessen werden die einzelnen, für die Codeverwaltung zuständigen Objekte in **CompilerGlobals** (**Tasks**, **Subs**), in der für NBC gewünschten Reihenfolge durchlaufen und liefern ihren Teil des Zielcodes. Die Verwaltung der Erzeugung übernimmt eine eigene Klasse **CodeGenerator**, diese verwaltet auch die nötigen Zähler.
Eine genaue Übersicht über die Sprache NBC findet sich unter [NBCDOC (2006)].

[12] In NNQC dürfen Identifier für Variablennamen nicht mit einem Unterstrich beginnen, in NBC schon. Lokale Variablen werden also nach einem Schema dieser Art abgebildet: _(funktions/task/subname)_variablenname.

Es werden Zähler zum Bestimmen der Schleifennummer benötigt, da sie für die Generierung der Sprungmarken im NBC-Code wichtig sind. Zum Speichern von Zwischenergebnissen wird ein globales Array verwendet. Die dort vorhandenen Speicherplätze werden reserviert und wieder freigegeben. Dazu werden entsprechende Methoden implementiert:

- *getNextLoopNum()* liefert die nächste, zu benutzende Schleifennummer für eine bestimmte Art von Schleife. Die Art wird als Parameter übergeben.
- *getFreeStackPlace()* liefert einen freien Platz im Array für Zwischenergebnisse (als eine Art Registerersatz).
- *freeStackPlace(***int num***)* gibt den benutzten Speicherplatz wieder frei.

Eine Übersicht über die unterschiedliche Codeerzeugung bieten die nächsten Abschnitte.

3.17.1. Direkte Übersetzung von Statements

Für die Befehle *strcat* , *syscall*, *exit*, *wait*, *set*, *get* und *numtostr* sind die Übersetzungen trivial, der Aufruf kann direkt in NBC umgesetzt werden. Beispielsweise führt ein *strcat(zielvar, var1, var2);* dann zu *strcat zielvar, var1, var2* in NBC.

3.17.2. Tasks und Subroutinen

Bei Tasks und Subroutinen beginnt die Coderzeugung mit der Erzeugung des entsprechenden Kopfes, danach folgt die Abarbeitung der gespeicherten Statements.

3.17.3. Funktionsaufrufe

Funktionen bilden eine Besonderheit, da es sich um Inline-Funktionen handelt, also an die Stelle im NNQC-Sourcecode der Code der Funktion hineinkopiert wird. Zur Realisierung der Funktionen in NBC gibt es grundsätzlich zwei Möglichkeiten:

- Realisierung über Sub(routine). Dabei werden die Parameter über eine globale Struktur übergeben. Nachteil ist hier die Verringerung der möglichen Subroutinen, aufgrund der Gesamtbeschränktheit auf 255.
- Benutzung von Inline-Code. Dabei gibt es keine Beschränkungen auf eine maximale Anzahl der Funktionen. Je nach Variablengebrauch (Erzeugen von lokalen Variablen), kann sich aber dort eine Beschränkung ergeben. Nachteil ist die Vergößerung des Sourcecodes für jede Benutzung einer Funktion.

Bei NNQC kommt die zweite Variante zum Einsatz. Eine definierte Funktion wird nur dann in den Code eingefügt, wenn sie auch verwendet wird. Dadurch ergibt sich der Vorteil, dass die Definition einer Funktion nicht automatisch die Gesamtzahl der zur Verfügung stehenden Variablen verringert. Sollte sich dennoch die Notwendigkeit ergeben, Subroutinen ebenfalls mit Argumenten auszustatten, so ist dies nachträglich mit geringem Aufwand möglich.

Bei der Erzeugung eines Objektes vom Typ **Function** wird diesem die Liste seiner Parameter übergeben. Der Programmcode der Funktion wird vom Visitor angefügt. Eine Funktion *getCode*, die als Parameter die Namen der aktuellen Parameter enthält, liefert den entsprechenden NBC-Code. Dabei werden die Variablennamen im Funktionscode durch die entsprechenden des Funktionsaufrufs ersetzt, zumindest wenn bei der Funktionsdefinition ein "&" hinter der Variable benutzt wurde, um diese als Referenz/Zeiger zu kennzeichnen. Fehlt diese Markierung, so wird eine neue lokale Variable erzeugt (siehe Abschnitt 3.15).

3.17.4. Repeat-Schleifen

Im Folgenden ist der Pseudocode für die Repeat Schleife aufgeführt. Diese Schleife führt die angegebenen Statements so oft aus, wie der arithmethische Ausdruck dies vorgibt. Der Pseudocode für die Erzeugung in NBC lautet:

```
doRepeat(Arith_expression, Statements) :
    cp = getFreeStackplace
    n = getNextLoopNum(Contants.KW_REPEAT)
    do_arith(Arith_expression, cp)
    code ("REPEATnStart:")
    code("brtst LTEQ, REPEATnEND, _astack[cp]")
    do_statements(Statements)
    code("sub _astack[cp], _astack[cp], 1")
    code("jmp RLOOPnStart")
    code("REPEATnEnd:")
    freeStackPlace(cp)
```

Die gewählte Form des Pseudocodes sollte weitgehend selbsterklärend sein, einige Konstrukte werden an dieser Stelle etwas näher betrachtet. *code(String)* bedeutet, dass der folgende String an den Code angehängt wird). Es gibt zwei weiterere genutzte Funktionen der Codeerzeugung, nämlich do_arith() und do_statements(). Diese dienen der Abarbeitung der arithmetischen Ausdrücke und Statements und der Ausgabe des entsprechenden NBC-Codes.
Die Namen der Sprungmarken werden anhand der aktuellen Schleifennummer gebildet, sie

sind dadurch im gesamten Zielcode eindeutig. Der erzeugte NBC-Code ist ähnlich wie Assembler, es gibt sowohl Labels (z. B. REPEATnStart), als auch Befehle wie *jmp* (Sprung zu einer Adresse) oder *sub* (Subtraktion). Der Befehl *brtst* mit dem Argument *LT* führt dazu, dass die folgenden Sprungmarke RLOOPnEND angesprungen wird, wenn der Wert von _astack[cp] kleiner oder gleich 0 sein sollte. Das benutzte freeStackPlace() dient dazu, den vorher belegten Platz im Zwischenspeicherarray wieder freizugeben.

3.17.5. Do-while und while-do-Schleifen

Der Code für while-do und do-while Schleifen unterscheidet sich vom Prinzip her nicht groß, es wird eine **compare_expression** statt einer arithmetischen benutzt. Der Pseudocode lässt sich im Anhang Kapitel A.4.1 einsehen.

3.17.6. Zuweisungen

Bei Zuweisungen muss der unterschiedliche Typ der Variablen auf der linken Seite, also letztendlich ob es sich um die Zuweisung eines Strings oder eines numerischen Wertes handelt, beachtet werden. Die Stringzuweisung ist der einfachste Fall, dabei wird das Argument (der zu setzende String) per **mov** in die Variable oder den Teil der Struktur geschrieben. Zuvor muss eventuell der Name (bei lokalen Variablen) aufgelöst und entsprechend verändert werden. Dies geschieht mit der Funktion getRealname(). Der Pseudocode lässt sich hier im Anhang Kapitel A.4.2 einsehen.

3.17.7. If-then-else

Im Falle von if-then-else lässt sich dieses mit if-then zusammenlegen. Es wird einfach immer die Marke für das entsprechende Else (markiert durch “IFnElse") angesprungen, wenn dort keine Statements folgen, geht es gleich weiter zur End-Marke (siehe A.4.3).

3.17.8. switch-case

Bei switch-case-Anweisungen ist zu bedenken, dass eventuell mehrere case-Bereiche zu durchlaufen sind, wenn kein “break” den vorherigen Bereich abschließt. Break ist – soweit vorhanden – immer das letzte Kommando in einem case-Block. Sollte es vohanden sein, wird direkt an das Ende des switch-case Blockes gesprungen (Code ist im Anhang A.4.4).

3.17.9. Arithmetische Ausdrücke

Die Code-Erzeugung für Arithmetische Ausdrücke unterscheidet sich etwas von der der übrigen Anweisungen. Der Grund dafür ist das mögliche Vorhandensein von Klammern in diesen. Um eine große Anzahl von (temporären) Speicherplätzen zum Zwischenspeichern der Ergebnisse von Klammern einzusparen, empfiehlt es sich, die Arithmetischen Ausdrücke vom Infix- in Postfix-Notation (UPN) umzuschreiben. Dadurch werden die Klammern eliminiert. Der so neu entstandende Ausdruck kann mit Hilfe eines Stacks einfach in NBC-Code umgewandelt werden.

3.17.10. Vergleichsausdrücke

Vergleichsausdrücke bestehen aus einer beliebig langen Kette von Vergleichen, die durch die logischen Operatoren *UND* und *ODER* verknüpft werden können. Ein Vergleich besteht dabei aus zwei Arithmethischen Ausdrücken und einem Vergleichsoperator. Zur Auswertung werden mindestens drei temporäre Speicherplätze benötigt, und zwar je zwei für das Ergebnis der Arithmetik und einer für das Ergebnis des Vergleiches. Handelt es sich um eine Kette von Vergleichen, ist ein weiterer Speicherplatz zur Zwischenspeicherung des Ergebnisses des nächsten Vergleichs nötig.

4. Realisierung

Dieses Kapitel befasst sich mit der praktischen Realisierung des NNQC-Compilers. Dabei wird das vorgesehene Design implementiert. Vom Design abweichende Implementierungen werden in diesem Kapitel näher betrachtet, ebenso wie einige exemplarische Umsetzungen.

4.1. Ablaufsteuerung

Wie schon im Design geplant (siehe 3.16), erfolgt die Steuerung durch eine Klasse **Main**. Diese sorgt dafür, dass die die Schichten repräsentierenden Objekte und Methoden in der richtigen Reihenfolge (siehe Abbildung 3.3) sequentiell aufgerufen werden.

4.2. Präprozessor

4.2.1. Aufruf des Präprozessors

Der Aufruf des Präprozessors erfolgt direkt aus **Main** des Compilers heraus, er ist somit direkt eingebunden. Eventuelle Fehlermeldungen sind, wie auch später beim Compiler, über die **Errorlist** in den CompilerGlobals abrufbar.

4.2.2. Include-Zyklen

Die Zyklenerkennung (siehe 3.9.5) wurde aus Zeitgründen nicht implementiert.

4.3. Compiler

4.3.1. Probleme mit der Fehlerbehandlung

Wie sich bei der Implementation herausstellte, unterstützt der mit JTB generierte Parser kein Error Recovery.
Dies ist eindeutig eine Schwachstelle von JTB, es führt dazu, dass der Compilerlauf nach dem ersten Auftreten einer Parse-Exception von Scanner oder Parser abgebrochen werden muss. Die Ausgabe einer Fehlermeldung ist natürlich möglich, allerdings wird nur der erste Fehler gefunden. Dies gilt nicht für Fehler, die von den Visitoren gefunden werden (Deklaration-, Definitions-, Typfehler), da dafür keine Exceptions zum Einsatz kommen.

4.3.2. Visitoren

Die Umsetzung der Visitoren aus dem Design erfolgte weitgehend wie geplant. Abweichungen und Ergänzungen sind in den folgenden Absätzen zu finden. Ein Klassendiagramm der benutzten Visitoren findet sich in Abbildung 4.1.

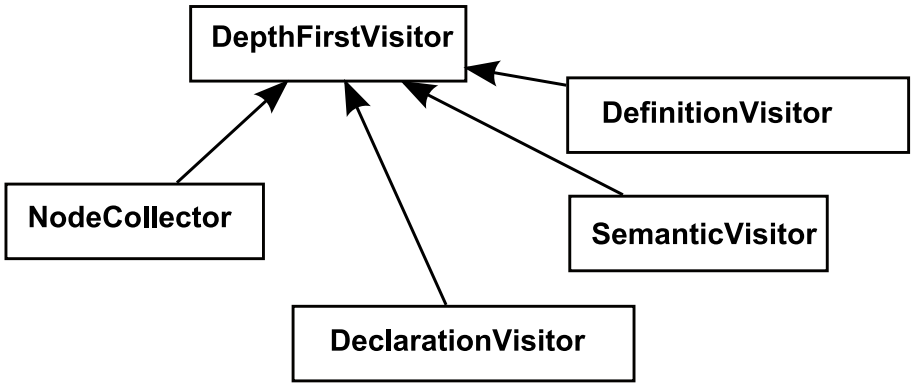

Abbildung 4.1.: Visitor Klassenhierarchie

NodeCollector

Die Klasse **NodeCollector** wird benutzt, um die Blätter eines Teilbaumes zu sammeln, dabei können bestimmte Knotentypen (definiert über den int-typ oder einen passenden String) ausgenommen werden. Dieser Visitor findet immer dort Anwendung, wo alle Blätter gesammelt werden sollen, und verhindert dort eine ganze Reihe von sonst nötigen Typecasts. Er liefert eine Liste von **Node** Objekten zurück (**NodeList**), die aus der Tiefentraversierung entstanden ist. Da es sich somit um Blätter des Baumes handelt, ist ein Typecast auf **NodeToken** – und damit Zugriff auf Typ und Stringrepräsentation – möglich.

DefinitionVisitor

Mit Hilfe dieses Visitors werden die Definitionen von Strukturen, sowie Subroutinen, Tasks und Funktionen gesammelt. Für letztere werden entsprechende neue Objekte erzeugt und diese in den **CompilerGlobals** gespeichert.

DeclarationVisitor

Dieser Visitor sammelt die Deklarationen von Variablen und Strukturen. Zudem ist die Funktion des CodeCollectors (siehe 3.2.2) integriert, da es sich um lediglich eine zusätzliche *visit*-Methode handelt. Da die aktuelle Subroutine im Visitor bekannt ist, und über **CompilerGlobals** der Zugriff auf die schon erzeugten Objekte für diese besteht, hängt die *visit*-Methode für **statement** die einzelnen Statements an dortige Liste an. Lokal deklarierte Variablen werden dort ebenfalls hinzugefügt.

SemanticVisitor

Der Visitor zur semantischen Überprüfung deckt die folgenden Fälle ab:

- Typüberprüfung bei Zuweisungen.
- Typüberprüfungen bei den Funktionen *strcat* und *numtostr*.
- Überprüfung der Parameter bei Funktionsaufrufen auf Anzahl und Typ.

Eine weitere semantische Prüfung findet derzeit nicht statt.

4.3.3. Codegenerierung

Der Generator für den NBC-Sourcecode ist nicht als Visitor implementiert. Der Hauptgrund für diese Vorgehensweise ist, eine Vergleichsmöglichkeit zu den Erfahrungen mit Visitoren und JavaCC/JTB zu sammeln. Der Zugriff auf die Knoten des Baumes erfolgt hauptsächlich mit Hilfe von Typecasts und der Benutzung des **NodeCollector**-Visitors.
Der **CodeGenerator** verfügt über eine zentrale Methode zum Aufruf von weiteren Methoden für die Behandlung der unterschiedlichen Anweisungen. Innerhalb dieser wird direkt auf die Bestandteile der einzelnen Nodes zugegriffen, dazu werden Typecasts benutzt. Im Vergleich mit den bisher benutzen Visitoren zeigt sich, dass die Anzahl der Typecasts relativ hoch ist (teilweise werden drei zum Zugriff auf ein Blatt benötigt), und damit auch fehleranfällig[1].
Die Erzeugung des NBC-Zielcodes wird dabei in folgenden Schritten durchgeführt:

[1] Fehler beim typecasting treten erst zur Laufzeit als ClassCassException auf.

1. Definition von verwendeten Strukturen.
2. Deklaration von globalen Variablen.
3. Deklaration von lokalen Variablen mit entsprechendem Namensschema als globale Variablen. Dazu werden die vorhandenen Funktionen, Subroutinen und Tasks durchlaufen, um die Liste an lokalen Variablen zu erhalten (siehe 3.15).
4. Deklaration von intern benutzten Variablen (temporäre Variablen für das Speichern von Zwischenergebnissen).
5. Durchlaufen der Liste von Tasks, Erzeugung des Kopfes, danach Abarbeiten der gespeicherten **statements**.
6. Durchlaufen der Liste von Subroutinen, Erzeugung des Kopfes, danach Abarbeiten der gespeicherten **statements**.

Die einzelnen Schritte speichern den von ihnen erzeugten Code in Strings, die später zum gesamten Code zusammengesetzt werden. Zur Verarbeitung von arithmetischen Ausdrücken werden dabei Hilfsklassen eingesetzt. Diese bringen die ihnen übergebene Liste von Operatoren und Operanten in die Form der UPN, bevor der NBC Code erzeugt wird.
Die Implementation der Methoden für die einzelnen Anweisungen entspricht im Wesentlichen den im Design entwickeltem Pseudocode (siehe A.4).
Abweichungen vom Design (vergleiche 3.17.3) gibt es bei der Codegenerierung für Funktionen. Anstatt im Sourcecode eine Ersetzung der Namen für die benutzten Variablen vorzunehmen, werden lokale Variablen in der Funktion benutzt. Die als Argumente übergebenen Werte in Form von Variablen oder Literale werden in diese lokalen Variablen kopiert. Am Ende der Funktion werden die Ergebnisse dann – soweit bei der Definition der Funktion ein Parameter als call-by-reference (mit "&") markiert wurde – wieder auf die Ausgangsvariablen zurückkopiert.
Eine weitere Abweichung besteht darin, dass die Subroutinen (also Funktionen, Subs und Tasks) den Zielcode nicht selber generieren, lediglich bei Funktionen wird der generierte Code zwischengespeichert. Der Grund dafür ist, dass er voraussichtlich mehrfach vorkommt.
Das zum Speichern von Zwischenergebnissen verwendete Array muss vor der Benutzung mit der NBC-Funktion *arrinit* initialisiert werden. Ansonsten kommt es zu Laufzeitfehlern auf dem NXT.

4.3.4. Aufruf von NBC

Der Aufruf von NBC erfolgt über eine eigene Klasse. Der Pfad zum Aufruf des NBC Compilers muss über einen Kommandozeilenparameter an den NNQC-Compiler übergeben werden. Das angegebene Programm muss eventuelle Kommandozeilenparameter von NBC akzeptieren, es ist aber auch möglich, ein Script oder anderes Programm als Wrapper zu be-

Tabelle 4.1.: Kommandozeilenparameter

Parameter	zusätzliches Argument	Funktion
-N	Absoluter Pfad zur NBC-Exe	vollständiger Pfad zum NBC-Compiler
-I	Absoluter Pfad	Pfad zum Auffinden der include-Dateien
-O	Absoluter Pfad	Pfad zur Bytecode-Datei
-T	Absoluter Pfad	Pfad zur NBC-Datei

nutzen. Damit ist es unter Linux beispielsweise einfach möglich ein Script zu benutzen, das NBC über wine aufruft.

4.3.5. Unterstütze Kommandozeilenparameter

Tabelle 4.1 zeigt die vom NNQC-Compiler unterstützten Parameter während des Programmaufrufs. Zur Implementierung wird CLI vom Apache/Jakarta Projekt [CLI (2006)] verwendet. Dieses Framework bietet eine einfache Schnittstelle zum Parsen und Überprüfen von Kommandozeilenparametern.

Die NBC-Datei enthält den endgültigen NBC-Code vor dem Aufruf des (externen) Übersetzers. Durch Angabe aller vier Parameter ist eine Benutzung des NNQC-Compilers unter Windows und Linux problemlos möglich.

4.4. Tests

Die durchgeführten Tests sind nicht ausreichend, um eine fundierte Aussage zur (relativen) Fehlerfreiheit der Compilersoftware geben zu können. Einige zum Testen benutzte Beispielprogramme finden sich im Anhang (B).

4.5. Standardbibliothek

Um NNQC sinnvoll einsetzen zu können, ist der Aufbau einer Bibliothek für oft benötigte Funktionen notwendig. Als Beispiel wurden für diese Bibliothek die folgenden Funktionen implementiert.

4.5.1. print

Die Funktion *print(string out)* erwartet als Argument einen String bzw. eine Variable/Teil einer Struktur vom Typ String. Dieser String wird dann auf dem Display des LEGO Mindstorms NXT ausgegeben. Zur Anzeige von Zahlen sind diese mit *numtostr* manuell umzuwandeln. Zur Benutzung von *print* ist die Datei “print.h” per *include*-Direktive einzubinden.

4.5.2. get_ultra_sensor(int value&)

Diese Funktion schreibt den aktuell gemessenen Wert des Ultraschall-Sensors in die als Parameter übergebene Variable. Die Ausführung dauert etwas, da der Ultraschall-Sensor über den I^2C-Bus angesprochen werden muss. Der Code befindet sich in der Datei “get_ultra_sensor.h”.

4.5.3. start_motor(int motor, int speed)

start_motor wird benutzt, um einen der drei Motoren zu starten und ihn mit der als Argument angegebenen Geschwindigkeit (in Prozent) fahren zu lassen. Eine Geschwindigkeit von 0 kann benutzt werden, um den Motor zu stoppen. Zur Benutzung von *wait* ist die Datei “start_motor.h’ einzubinden.

4.6. Erweiterungsmöglichkeiten

Hier findet sich eine Sammlung von Ideen und Vorschlägen für mögliche Erweiterungen des NNQC-Compilers.

4.6.1. Eclipse Plugin

Die Entwicklung eines Plugins für eine bestehende Entwicklungsumgebung wäre sinnvoll. Durch Techniken wie Syntaxhighlighting und Analyse des Codes durch das Plugin, mit dem Aufzeigen von Problemen für die AnwenderInnen, würde die Arbeit mit NNQC deutlich erleichtert. Das Plugin könnte zudem die Möglichkeit bieten, die Software direkt über Bluetooth oder USB auf den LEGO Mindstorms NXT zu übertragen. Auch die Möglichkeit von direktem Debugging wäre eine interessante Funktion, diese würde jedoch wahrscheinlich Änderungen am NXT-Betriebssystem erfordern. Um eine bessere Einbindung des Compilers in solch ein Plugin zu erzielen, wären Änderungen an der Klasse **Main** erforderlich,

oder das Neuschreiben einer eigenen Methode für die Ablaufsteuerung des Compilerlaufs. Die Auswertung der Fehlermeldungen müsste zu einem Markieren der entsprechenden Stellen im Sourcecode führen, dies sollte allerdings recht einfach vorzunehmen sein.

4.6.2. Codeoptimierung und Zwischencode

Der derzeit erzeugte NBC-Code ist nicht optimiert, er wird eins zu eins aus dem NNQC-Code erzeugt. Es gibt zahlreiche Möglichkeiten, den Code zu optimieren, beispielsweise wäre das Entfernen von unerreichbarem Code sinnvoll. Auch die Informierung des Benutzers über Programmcode in diesem Fall, wäre zu begrüßen. Um diese Optimierungen einfach vornehmen zu können, wäre die Erstellung eines unabhängigen Zwischencodes, beispielsweise in Form eines nicht (streng) typisierten Baumes notwendig. Dies würde auch die Verwendung einer anderen Quellsprache für das Frontend deutlich erleichtern.

4.6.3. Erweiterungen der Sprache

Unterstützung von Arrays

Die Unterstützung von Arrays innerhalb von NNQC wäre eine sinnvolle Erweiterung. Dies würde die Möglichkeit bieten, Funktionen für eine Stackverwaltung zu erzeugen, ebenso wie die Möglichkeit, eine größere Menge an Daten in Variablen zu speichern, als dies jetzt (bedingt durch die Begrenzung auf 255 Variablen insgesamt) der Fall ist.

Erweiterungen bei Funktionen und Subroutinen/Tasks

Zur einfacheren Handhabung und zur Vollendung der Kapselung sollten Funktionen und Subroutinen über Rückgabewerte verfügen. Subroutinen sollten über Parameter verfügen. Dabei müsste besonderes Augenmerk auf die Behandlung von Rekursionen gelegt werden, die dabei möglich sind.Für die Parameter müsste so zwingend ein Stack benutzt werden, um das Überschreiben der verwendeten globalen Variablen zu verhindern.
Alle Erweiterungen an NNQC führen notwendigerweise zu Änderungen in der Grammatik. Anpassungen der Visitoren sind daraus eine Folge.

Unterstützung von Scheduling

Die Verwendung mehrerer Tasks und ein Scheduling derselben wäre eine weitere sinnvolle Erweiterung. Dabei wäre insbesondere die Frage abschließend zu klären, ob die bestehenden

Einschränkungen aus den Fähigkeiten der Firmware resultieren, oder aus der Implementation von NBC.

4.6.4. Direkte Ausgabe von Bytecode

Nach der Freigabe des Sourcecodes für die Firmware, beziehungsweise der Freigabe der Dokumentation des Bytecodes, wird es möglich sein, den Umweg über NBC zu vermeiden. Der Bytecode kann dann vom Compiler direkt erzeugt werden. Inwieweit dann Änderungen an der Struktur der Codegenerierung beziehungsweise des Zwischencodes notwendig sind, wird sich erst dann herausstellen. LEGO hat Anfang August 2006 die Dokumentation des Bytecode- und Dateiformats für den LEGO Mindstorms NXT ([NXTBCDOC (2006)]) freigegeben. NQC ist danach (wie erwartet) eine fast direkte Umsetzung des Funktionsumfanges der Firmware und syntaktisch sehr nah am Bytecode.

4.7. Bewertung

Die Kombination aus JavaCC und JTB ist gut geeignet zur Erstellung eines Scanner/Parser Duos für den Compilerbau. Es fielen jedoch eindeutige Schwachstellen auf:

- Der (durch Typisierung) starre Syntaxbaum verhindert die direkte Verwendung als Zwischencode, da er kaum Änderungen zulässt. Zusätzlich erfordert diese Typisierung ständige Typecasts beim Zugriff auf optionale Elemente oder Wahlmöglichkeiten.
- Die fehlende Unterstützung von Error Recovery durch die generierten Parser/Scanner.

Die Benutzung einer LL(1) Grammatik und der Aufbau des Syntaxbaumes erwiesen sich als eindeutiger Vorteil, Probleme innerhalb der Grammatik waren schnell aufzufinden. Die Einarbeitungszeit in JavaCC/JTB hielt sich, insbesondere aufgrund der durch die vorbereitende Vorlesung “Compiler und Interpreter” vorhandenen Kenntnisse des Autors, in einem erträglichen Rahmen. Im Nachhinein wäre aber vielleicht eine tiefergehende Suche nach alternativen Metacompilern sinnvoll gewesen, insbesondere SableCC ([SABLECC (2006)]) scheint ein vielversprechender Kandidat zu sein, der beispielsweise Unterstützung für Error Recovery beziehungsweise. Panikmodus bietet.
JavaCC hat die qualitativen Erwartungen insgesamt erfüllt, es gab während der gesamten Entwicklungszeit keine ernsthaften Probleme mit der Software, sie funktionierte fehlerfrei. JTB würde ein bisschen mehr an aktueller Dokumentation sicherlich gut tun.
Die Entscheidung, keinen unabhängigen Zwischencode zu erzeugen, war, was die Geschwindigkeit und den Aufwand anging, richtig. Das Fehlen eines unabhängigeren Zwischencodes ist für zukünftige Erweiterungen jedoch von Nachteil.

Die Entscheidung auf Basis von NQC eine eigene Sprache und damit notwendigerweise eine eigene Grammatik zu entwickeln führte zu einem erheblichen zeitlichen Aufwand. Ob dieser Aufwand größer oder kleiner als die Einarbeitung in andere Verfahren der Compilergenerierung, wie die Benutzung der existierenden Lex/Yacc-LALR-Grammatik für NQC ist, lässt sich natürlich nicht wirklich beurteilen, ohne diesen Schritt vergleichsweise durchzuführen.
Der als Prototyp enstandene Compiler ist in der Lage, lauffähige NNQC-Programme für den LEGO Mindstorms NXT zu übersetzen, er bietet somit eine gute Basis für zukünftige Arbeiten in diesem Bereich.

5. Schluss

Dieses Kapitel soll dazu dienen, einen Rückblick und eine Bewertung dieser Arbeit zu geben.

5.1. Rückblick und Bewertung

Ziel der Arbeit war die Entwicklung einer C-ähnlichen Programmiersprache und eines entsprechenden Compilers.
Dazu wurden zunächst die Anforderungen an eine solche Sprache betrachtet, ebenso wie die Bedingungen, die durch die Hard- und Software des Zielsystems gegeben sind. Daraus wurde der Schluss gezogen, dass die einzige Möglichkeit, Code für den LEGO Mindstorms NXT zu erzeugen, in der Benutzung von "Next Byte Code" als Zielsprache besteht.
Als syntaktische Grundlage für die zu erstellende Sprache wurde "Not Quite C" gewählt, der dafür zu erstellende Compiler sollte in Java entstehen. Nach einer Evaluierung verfügbarer Werkzeuge für die Erstellung von Compilern unter Java fiel die Wahl dann auf die Kombination JavaCC/JTB.
In der folgenden Phase wurden dann zunächst die Einflüsse von JavaCC/JTB auf das Design betrachtet. Es wurde die generelle Architektur des Compilers entworfen, gefolgt von einer genauen Spezifikation der NNQC-Programmiersprache. Nach der Feststellung, dass ein Präprozessor eine weitere sinnvolle Ergänzung zum Compilerentwurf darstellen würde, folgte ein Entwurf für einen solchen.
Es folgte die Beleuchtung weiterer Aspekte wie die Unterschiede zu NQC, der Fehlerbehandlung und der Möglichkeit Funktionsbibliotheken für die Sprache zu erstellen. Die Phase des Designs wurde schließlich mit der Erstellung von Pseudocode für die Codegenerierung abgeschlossen.
Danach folgte die Implementierung eines Compilers als Prototyp, der weitestgehend dem Design folgte. Vorhandene Änderungen und Abweichungen wurden kurz erläutert. Der entstandene Compiler verfügt dabei im Großen und Ganzen über die gewünschte Funktionalität, im Bereich der semantischen Analyse und Fehlererkennung gibt es allerdings noch einige Lücken. Ein Ausblick auf zukünftige Möglichkeiten der Erweiterung und eine Bewertung der verwendeten Werkzeuge schloss die Phase der Realisierung ab.

Insgesamt lässt sich zusammenfassen, dass das Ziel, einen Compiler für eine Programmiersprache mit C-ähnlicher Syntax zu entwickeln, erreicht wurde. Der im Laufe dieser Arbeit entwickelte Compiler sollte eine gute Basis für eine weitere Entwicklung bieten, insbesondere die Erweiterung der NNQC-Standardbibliothek und der Dokumentation können auch von anderen als dem Autor dieser Arbeit durchgeführt werden. Dazu bietet sich die Freigabe des Codes unter einer Open-Source-Lizenz an.

Durch die Freigabe des Compilers und des Sourcecodes unter einer Open-Source-Lizenz, sollten sich die hauptsächlich noch bestehenden Probleme Dokumentation, Testen und Erweiterung der Standardbibliothek mit Hilfe von anderen lösen lassen. Durch die anstehende weitere Verbreitung des LEGO Mindstorms NXT, sollte auch eine geeignete Basis für das Entstehen einer Community vorhanden sein. Die Hardware bietet eine gute Voraussetzung für eine weitgehende Verbreitung des NXT. Durch die schon integrierten Erweiterungsmöglickeiten, ist mit einer schnellen Verfügbarkeit weiterer Sensoren oder anderer Hardware von Fremdherstellern zu rechnen, die dann auch ohne Wechsel der Firmware sofort zum Einsatz kommen können. Dies zusammen bietet die Grundlage, aus der sich ein Erfolg des NXT und zahlreiche Erweiterungen der Hard- und Software ergeben können.

5.2. Fazit

Der bei dieser Arbeit eingeschlagene Weg hat sich insgesamt als richtig erwiesen. Dennoch bleiben Zweifel, ob er optimal war. Bei einer erneuten ähnlichen Problemstellung würde der Autor deshalb weitere Alternativen im Bezug auf den verwendeten Metacompiler in Betracht ziehen und aufgrund seiner Erfahrungen neu bewerten.
Dabei wären die beiden problematischen Stellen der Kombination JavaCC/JTB (Error Recovery und Flexibilität des Syntaxbaumes) natürlich Punkte von besonderem Interesse.
Die Entwicklung des NNQC-Compilers kann nur ein Schritt in Richtung eines umfassenden Systems zur freien Programmierung des NXT mit einer textbasierten Programmiersprache darstellen. Sie ist allerdings ein wichtiger Schritt.
Eine wirkliche Relevanz wird NNQC nur erreichen, wenn die Software benutzt und weiterentwickelt wird.

Der Autor hofft, mit der entwickelten Software einen kleinen Beitrag geleistet zu haben, um den LEGO Mindstorms NXT als Plattform zu etablieren, die jungen Menschen einen einfachen Einstieg in die Welt der Informatik ermöglicht.

Literaturverzeichnis

[ALPHAREX 2006] A/S, LEGO: *Lego Nxt AlphaRex*. Verifiziert am 4.08.2006. 2006. – URL http://cache.lego.com/upload/contentTemplating/LEGOAboutUs-ImageLibrary/images/2057/picFDB5DB07-8A2A-4B77-B231-DB49546F007C.jpg

[BSDLIC 1998] DIVERSE: *Open Source Initiative OSI - The BSD License:Licensing*. Verifiziert am 14.07.2006. 1998. – URL http://www.opensource.org/licenses/bsd-license.html

[CLI 2006] FOUNDATION, Apache S.: *Home - Commons CLI*. Verifiziert am 10.08.2006. 2006. – URL http://jakarta.apache.org/commons/cli/

[CUP 2006] HUDSON, Scott: *CUP Parser Generator for Java*. Verifiziert am 6.08.2006. 2006. – URL http://www.cs.princeton.edu/~appel/modern/java/CUP/

[DRACHENBUCH 1999] AHO ; SETHI ; ULLMANN: *Compilerbau*. Oldenbourg, 1999

[GUETING 1999] GÜTING, Erwig: *Übersetzerbau: Techniken, Werkzeuge, Anwendungen*. Heidelberg, 1999

[HELMICH 2006] HELMICH, Ulrich: *Compilerbau*. Verifiziert am 14.07.2006. 2006. – URL http://www.u-helmich.de/inf/comp/index.html

[HITECHNIC 2006] INC, Hitechnix: *HiTechnic Products*. Verifiziert am 4.08.2006. 2006. – URL http://www.hitechnic.com

[JAVACC 2006] DIVERSE: *JavaCC Home*. Verifiziert am 14.07.2006. 2006. – URL http://javacc.dev.java.net

[JAVACCECLIPSE 2006] ROUTCHERAWY, Remi: *JavaCC Eclipse Plugin*. Verifiziert am 14.07.2006. 2006. – URL http://perso.orange.fr/eclipse_javacc/

[JAVACOMPILER 2006] DIVERSE: *Java Compiler Tools*. Verifiziert am 14.07.2006. 2006. – URL http://catalog.compilertools.net/java.html

[JAVAPATTERNS 1998] GRAND, Mark: *Patterns in Java, Volume 1*. Wiley and Sons, 1998

[JAY 2006] SCHREINER, Axel T. ; KÜHL, Bernd: *jay - Ein yacc für Java*. Verifiziert am 3.08.2006. 2006. – URL http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/staff/design/de/Artikel.htmld/index.html

[JFLEX 2006] KLEIN, Gerwin: *JFlex - The Fast Scanner Generator for Java*. Verifiziert am 12.07.2006. 2006. – URL http://www.jflex.de

[JJTREEDOC 2006] VANTER, Michael Van D.: *JavaCC: JJTree Reference*. Verifiziert am 14.07.2006. 2006. – URL https://javacc.dev.java.net/doc/JJTree.html

[JLEX 2006] BERK, Elliot J. ; ANANIAN, C. S.: *JLex: A Lexical Analyzer Generator for Java*. Verifiziert am 12.07.2006. 2006. – URL http://www.cs.princeton.edu/~appel/modern/java/JLex/

[JTB 2006] WANG, Wanjun: *JTB: The Java Tree Builder Homepage*. Verifiziert am 3.08.2006. 2006. – URL http://compilers.cs.ucla.edu/jtb/jtb-2003/

[JTBJJTREE 2006] DIVERSE: *Other Tools*. Verifiziert am 14.07.2006. 2006. – URL http://compilers.cs.ucla.edu/jtb/jtb-2003/tools.html

[JVERNE 2003] AGENCY, European S.: *ESA - ATV - Jules Verne: the first and most elaborate space rendezvous for Europe*. Verifiziert am 10.08.2006. 2003. – URL http://www.esa.int/SPECIALS/ATV/SEMJXMS1VED_0.html

[LEGONXT 2006] A/S, LEGO: *Lego Nxt*. Verifiziert am 4.08.2006. 2006. – URL %http://cache.lego.com/upload/contentTemplating/LEGOAboutUs-PressReleases/images/2057/picAA00C400-D0CE-4118-8762-82A18499E875.jpg

[LEPOMUX 2006] UNBEKANNT: *Lepomux Port Multiplexer*. Verifiziert am 13.07.2006. 2006. – URL http://www.lepomux.org

[MSROBOTICS 2006] CORPORATION, Microsoft: *Microsoft Robotics Studio*. Verifiziert am 19.08.2006. 2006. – URL http://msdn.microsoft.com/robotics/

[NBCDOC 2006] HANSEN, John: *Next Byte Code*. Verifiziert am 25.07.2006. 2006. – URL http://bricxcc.sourceforge.net/nbc/

[NQC 2005] HANSEN, John: *NQC - Not Quite C*. Verifiziert am 12.07.2006. 2005. – URL http://bricxcc.sourceforge.net/nqc/

[NXTBCDOC 2006] A/S, LEGO: *mindstorms nxt'reme*. Verifiziert am 10.08.2006. 2006. – URL http://mindstorms.lego.com/Overview/nxtreme.aspx

[NXTSPEC 2006] A/S, LEGO: *The NXT*. Verifiziert am 4.08.2006. 2006. – URL http://mindstorms.lego.com/Overview/The_NXT.aspx

[ROBOMOWER 2006] INC, System Trade C.: *Automatic Lawn Mowers by Friendly Robotics*. Verifiziert am 4.08.2006. 2006. – URL http://www.friendlyrobotics.com/

[ROOMBA 2006] GMBH, Roombastic: *Romba CH- Romba SE- denn sie haben besseres zu tun*. Verifiziert am 4.08.2006. 2006. – URL http://www.roomba.ch/

[SABLECC 2006] DIVERSE: *SableCC parser generator*. Verifiziert am 10.08.2006. 2006. – URL www.sablecc.org

[VOELLER 2006] VÖLLER, Reinhard: *Formale Sprachen und Compiler*. Verifiziert am 14.07.2006. 2006. – URL http://users.informatik.haw-hamburg.de/~voeller/fc/comp/comp.html

[wine 2006] DIVERSE: *Wine HQ*. Verifiziert am 12.07.2006. 2006. – URL http://www.winehq.com/

[WIRTH 1995] WIRTH, Niklaus: *Grundlagen und Techniken des Compilerbaus*. Eddison-Wesley, 1995

[WPCOMPILERBAU 2006] DIVERSE: *Wikipedia:Compilerbau*. Verifiziert am 14.07.2006. 2006. – URL http://de.wikipedia.org/wiki/Compilerbau

A. Quellcode

A.1. NNQC BNF Grammatik

```
kw_task = "task";
kw_function = "void" ;
kw_sub = "sub" ;
kw_exit = "exit" ;
kw_else = "else";
kw_if = "if";
kw_while = "while";
kw_do = "do" ;
kw_repeat = "repeat" ;
kw_switch = "switch" ;
kw_case = "case" ;
kw_default = "default" ;
kw_syscall = "syscall" ;
kw_struct = "struct" ;
kw_gettick = ``gettick'' ;
kw_int = "int" ;
kw_slong = "slong";
kw_ulong = "ulong";
kw_uword = "uword";
kw_sword = "sword";
kw_ubyte = "ubyte";
kw_sbyte = "sbyte";
kw_break = "break";
kw_return = "return"
kw_continue := "continue"
kw_default := "default"
kw_abs := "abs"
kw_sign := "sign"


op_eq = "==" ;
op_gt = ">" ;
op_lt = "<" ;
op_gteq = ">=" ;
op_lteq = "<=" ;
op_incr = "++" ;
op_decr = "\-\-" ;
as_norm = "=" ;
as_add = "+=" ;
as_sub = "\-=" ;
as_mul = "*=" ;
as_div = "/=";


INPUT = ( VARIABLE_DECLARATION | STRUCT_DECLARATION_OR_DEFININITON | TASK_DECLARATION |
          FUNCTION_DECLARATION | SUB_DECLARATION ) INPUT_R <EOF>
```

```
INPUT_R = ( VARIABLE_DECLARATION | STRUCT_DECLARATION_OR_DEFININITON | TASK_DECLARATION |
          FUNCTION_DECLARATION | SUB_DECLARATION ) INPUT_R | ϵ
VARIABLE_DECLARATION = VARTYPE VARNAME  VARIABLE_DECLARATION_R ";"
VARIABLE_DECLARATION_R = "," VARNAME VARIABLE_DECLARATION_R | ϵ
VARNAME = identifier
LOCAL_VARIABLE_DECLARATIONS = LOCAL_VARIABLE_DECLARATION LOCAL_VARIABLE_DECLARATIONS_R
LOCAL_VARIABLE_DECLARATIONS_R = LOCAL_VARIABLE_DECLARATION LOCAL_VARIABLE_DECLARATIONS_R |
          ϵ
LOCAL_VARIABLE_DECLARATION = VARIABLE_DECLARATION | STRUCT_DECLARATION ";" | ϵ
TASK_DECLARATION = kw_task identifier "()" "{"  LOCAL_VARIABLE_DECLARATION  STATEMENTS "}"
FUNCTION_DECLARATION = kw_function  identifier "("  FUNCTION_ARGLIST  ")" "{"
          LOCAL_VARIABLE_DECLARATION  STATEMENTS "}"
SUB_DECLARATION = kw_sub  identifier "()" "{" LOCAL_VARIABLE_DECLARATIONS STATEMENTS "}"
FUNCTION_ARGLIST = FUNCTION_ARG FUNCTION_ARGLIST_R | ϵ
FUNCTION_ARGLIST_R = "," FUNCTION_ARG FUNCTION_ARGLIST_R | ϵ
FUNCTION_ARG = (VARIABLE_TYPE | kw_struct STRUCT_NAME) VARNAME ARG_O
ARG_O = "&"|ϵ
STRUCT_NAME = identifier
STATEMENTS = STATEMENT STATEMENTS_R
STATEMENTS_R = STATEMENT STATEMENTS_R | \ϵ
STATEMENT = ASSIGNMENT | IFLOOP | DOLOOP | REPEATLOOP | SWITCHLOOP | WHILELOOP | SYSCALL |
          SET | GET |EXIT | NUMTOSTR | STRCAT |GETTICK | SUBFUNCTIONCALL | VAR_INC_DECR
EXIT = kw_exit "()" ";"
GETTICK = kw_gettick "(" STRUCTPART_OR_VARNAME ")" ";"
GET = kw_get "(" STRUCTPART_OR_VARNAME "," (DIGIT|HEXDIGIT) ")" ";"
SET = kw_set "(" (DIGIT|HEXDIGIT)"," ARITH_EXPRESSION ")" ";"
STRUCTPART_OR_VARNAME = STRUCTPART | VARNAME
NUMTOSTR = kw_numtostr "(" STRUCTPART_OR_VARNAME "," (DIGIT|STRUCTPART_OR_VARNAME) ")" ";"
STRCAT = kw_strcat "(" STRUCTPART_OR_VARNAME "," (STRING_LITERAL|STRUCTPART_OR_VARNAME)
          STRCAT_R ")" ";"
STRCAT_R =  "," (STRING_LITERAL|STRUCTPART_OR_VARNAME) STRCAT_R | ϵ
SYSCALL = kw_syscall "(" (DIGIT|HEXDIGIT) "," STRUCTPART_OR_VARNAME ")" ";"
SUBFUNCTIONCALL = identifier "(" SUBFUNCTIONCALL_O ")" ";"
SUBFUNCTIONCALL_O = ( ARITH_EXPRESSION| STRING_LITERAL ) SUBFUNCTIONCALL_O_R
SUBFUNCTIONCALL_O_R = "," ( ARITH_EXPRESSION| STRING_LITERAL ) SUBFUNTIONCALL_O_R | \epsilon
IFLOOP = kw_if "(" COMPARE_EXPRESSION ")" "{" STATEMENTS "}" IFLOOP_ELSE
IFLOOP_ELSE = kw_else "{" STATEMENTS "}" | ϵ
WHILELOOP = kw_while "(" COMPARE_EXPRESSION ")" "{" STATEMENTS "}"
REPEATLOOP = kw_repeat "(" ARITH_EXPRESSION ")" "{" STATEMENTS "}"
SWITCHLOOP = kw_switch "(" ARITH_EXPRESSION ")" "{" SWITCHLOOP_R DEFAULT "}"
SWITCHLOOP_R = kw_case DIGIT":" (STATEMENTS|kw_break ";") SWITCHLOOP_R | ϵ
DEFAULT = kw_default ":" (STATEMENTS|kw_break ";") | ϵ
DOLOOP = kw_do "{" STATEMENTS "}" kw_while "(" COMPARE_EXPRESSION ")"
VARIABLE_TYPE = kw_int|kw_string|kw_dword|kw_sword|kw_sbyte|kw_dbyte|kw_byte
COMPARE_EXPRESSION = ARITH_EXPRESSION (op_eq|op_gt|op_lt|op_gteq|op_lteq|op_neq)
          ARITH_EXPRESSION COMPARE_EXPRESSION_R
COMPARE_EXPRESSION_R = (OP_AND|OP_OR) COMPARE_EXPRESSION | ϵ
VAR_INCR_DECR = STRUCTPART_OR_VARNAME (op_incr|op_decr)";"
ASSIGNMENT = ( STRUCTPART_OR_VARNAME ) (as_norm|as_add|as_sub|as_mul|as_div)
          ARITH_EXPRESSION|string_literal ";"
STRUCTPART = VARIABLE "." identifier STRUCTPART_R
STRUCTPART_R = "." indentifier STRUCTPART_R |ϵ
ARITH_FUNCTION = op_abs|op_sign | ϵ
ARITH_EXPRESSION = ARITH_TERM ARITH_EXPRESSION_R
ARITH_EXPRESSION_R = (op_add|op_sub) ARITH_TERM ARITH_EXPRESSION_R | ϵ
ARITH_SIGN = op_add|op_sub|ϵ
ARITH_TERM = ARITH_FACTOR ARITH_TERM_R ;
ARITH_TERM_R = (op_mul|op_div|op_mod) ARITH_FACTOR ARITH_TERM_R | ϵ
ARITH_FACTOR =  ARITH_FUNCTION  "(" ARITH_EXPRESSION ")" | ARITH_SIGN (
          STRUCTPART_OR_VARNAME OP_INCR_DECR | DIGIT|HEXDIGIT)
OP_INCR_DECR = op_incr | op_decr | ϵ
STRUCT_DECLARATION_OR_DEFINITION = STRUCT_DEFINITION | STRUCT_DECLARATION ";"
```

```
STRUCT_DEFINITION = kw_struct identifier "{" (VARIABLE_TYPE | kw_struct identifier )
          identifier ";" STRUCT_DEFINITIO_R "}"
STRUCT_DEFINITION_R = (VARIABLE_TYPE | kw_struct identifier ) identifier ";"
          STRUCT_DEFINITION_R | ϵ
STRUCT_DECLARATION = kw_struct identifier VARIABLE VARIABLE_STRUCT_DECLARATION_R
STRUCT_DECLARATION_R = "," VARIABLE STRUCT_DECLARATION_R | ϵ
```

A.2. nnqc.jtb

Der Quellcode der Compiler/NNQC Grammatik für JavaCC/JTB.

```
// $Id: nnqc.jtb 195 2006-08-24 09:54:29Z arne $



options {
  LOOKAHEAD = 1;
  CHOICE_AMBIGUITY_CHECK = 2;
  OTHER_AMBIGUITY_CHECK = 1;
  STATIC = false;
  DEBUG_PARSER = false;
  DEBUG_LOOKAHEAD = false;
  DEBUG_TOKEN_MANAGER = false;
  ERROR_REPORTING = true;
  JAVA_UNICODE_ESCAPE = false;
  UNICODE_INPUT = false;
  IGNORE_CASE = false;
  USER_TOKEN_MANAGER = false;
  USER_CHAR_STREAM = false;
  TOKEN_MANAGER_USES_PARSER = true;
  BUILD_PARSER = true;
  BUILD_TOKEN_MANAGER = true;
  SANITY_CHECK = true;
  FORCE_LA_CHECK = false;
  OUTPUT_DIRECTORY = "nnqc";
}


PARSER_BEGIN(NnqcParser)
package nnqc;

import visitor.*;
import syntaxtree.*;


import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class NnqcParser {


}

PARSER_END(NnqcParser)

SPECIAL_TOKEN : /*comments*/
{
    <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
```

```
    |   <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/" >
}

SKIP: {" "|"\t"}
SKIP: {"\n"|"\r"|"\r\n"}



// operators
TOKEN: {<OP_ABS: "abs">
    |   <OP_SIGN: "sign">
    |   <OP_MUL:    "*">
    |   <OP_DIV:    "/">
    |   <OP_MOD:    "%">
    |   <OP_ADD:    "+">
    |   <OP_SUB:    "-">
    |   <OP_GT:     ">">
    |   <OP_LT:     "<">
    |   <OP_INCR:   "++">
    |   <OP_DECR:   "--">
    |   <OP_GTEQ:   ">=">
    |   <OP_LTEQ:   "<=">
    |   <OP_NEQ:    "!=">
    |   <OP_EQ:     "==">
    |   <OP_AND:    "&&">
    |   <OP_OR:     "||">
    |   <AS_NORM:   "=">
    |   <AS_ADD:    "+=">
    |   <AS_MUL:    "*=">
    |   <AS_SUB:    "-=">
    |   <AS_DIV:    "/=">
    |   <KOMMA:     ",">
}


// control/loops
TOKEN: {<KW_IF: "if">
    |   <KW_THEN:   "then">
    |   <KW_ELSE:   "else">
    |   <KW_WHILE:  "while">
    |   <KW_DO:     "do">
    |   <KW_FOR:    "for">
    |   <KW_REPEAT: "repeat">
    |   <KW_SWITCH: "switch">
    |   <KW_DEFAULT: "default">
    |   <KW_CASE:       "case">
    |   <KW_CONTINUE:   "continue">
    |   <KW_BREAK:  "break">
    |   <KW_SYSCALL: "syscall">
    |   <KW_RETURN: "return">
    |   <KW_TASK:       ("thread" | "task") >
    |   <KW_SUB:    "sub">
    |   <KW_FUNCTION:   "void">
    |   <KW_EXIT:       "exit">
    |   <KW_NUMTOSTR:   "numtostr">
    |   <KW_STRCAT:     "strcat">
    |   <KW_GETTICK:    "gettick">
    |   <KW_WAIT:       "wait">
    |   <KW_SET:        "set">
    |   <KW_GET:        "get">

}
```

```
// vars
TOKEN: {<KW_STRUCT:     "struct">
    |   <KW_INT:        ("int" | "sdword") >
    |   <KW_STRING:     "string">
    |   <KW_BYTE:       "byte">
    |   <KW_DWORD:      "dword">
    |   <KW_SWORD:      "sword">
    |   <KW_SBYTE:      "sbyte">
    |   <KW_DBYTE:      "dbyte">
}

TOKEN: {<CHARACTER_LITERAL: "\'" (~["\'","\\","\n","\r"] | "\\" (["n","t","b","r","f","\\",
           "\'","\""] | ["0"-"7"] (["0"-"7"])? | ["0"-"3"] ["0"-"7"] ["0"-"7"])) "\'">
    |   <STRING_LITERAL: "\"" ( ~["\"","\\","\n","\r"] | "\\" ( ["n","t","b","r","f","\\","
              \'","\""] | ["0"-"7"] (["0"-"7"])?  | ["0"-"3"] ["0"-"7"] ["0"-"7"] | ( ["\n
              ","\r"] | "\r\n")))* "\"">
    |   <HEXDIGIT: "0x" (["0" - "9", "a" - "f", "A" - "F"])+ >
    |   <IDENTIFIER: (["a"-"z","A"-"Z"])(["a"-"z","A"-"Z","0"-"9"])*>
    |   <DIGIT: (["0" - "9"])+>
    |   <RBRACKOPEN: "(">
    | <RBRACKCLOSE: ")">
}

// Main Input (root)
void Input(): {}
{
// reihenfolge spielt hier keine rolle...
(variable_declaration()| struct_declaration_or_definition()| task_declaration() |
          function_declaration() | sub_declaration())+ <EOF>
}


void variable_declaration() : {}
{
    variable_type() varname() (<KOMMA> varname())*  ";"
}


void varname(): {}
{
    <IDENTIFIER>
}

void local_variable_declarations(): {}
{
    (local_variable_declaration())+
}

void local_variable_declaration(): {}
{
    ( variable_declaration() | struct_declaration() ";")
}

void task_declaration(): {}
{
    <KW_TASK> <IDENTIFIER> "(" ")" "{" [ local_variable_declarations() ] statements() "}"
}

void function_declaration(): {}
{
    <KW_FUNCTION> <IDENTIFIER> "(" [ function_arglist() ] ")" "{" [
              local_variable_declarations() ] statements() "}"
}
```

```
void sub_declaration(): {}
{
    <KW_SUB> <IDENTIFIER> "(" ")" "{" [ local_variable_declarations() ] statements() "}"
}

void function_arglist(): {}
{
    function_arg() (<KOMMA> function_arg())*
}

void function_arg(): {}
{
    ( variable_type() | <KW_STRUCT> struct_name()) varname() ["&"]
}

// wird benutzt, um structs zu deklarieren,
// d.h. die struktur muss deklariert sein,
// wenn wir hier ankommen (analog zu variable)
void struct_name(): {}
{
    <IDENTIFIER>
}

void statements(): {}
{
    (statement())+
}

// lookahead, um assignment von subfunctioncall und var_incr_decr unterscheiden zu können (
          beides beginnt mit identifier)
void statement(): {}
{
     LOOKAHEAD(assignment()) assignment()|ifloop()|doloop()|repeatloop()|switchloop()|
                whileloop()|
    syscall()|stwait()|set()|get()|exit()|numtostr()|strcat()|gettick()|LOOKAHEAD(2)
               subfunctioncall()| var_incr_decr()

}

void exit(): {}
{
    <KW_EXIT> "(" ")" ";"
}

void gettick(): {}
{
    <KW_GETTICK> "(" structpart_or_varname() ")" ";"
}

void structpart_or_varname(): {}
{
    LOOKAHEAD(2) structpart()|varname()
}

void numtostr(): {}
{
    <KW_NUMTOSTR> "(" structpart_or_varname() <KOMMA> (<DIGIT>|<HEXDIGIT>|
               structpart_or_varname()) ")" ";"
}

void stwait(): {}
```

```
{
    <KW_WAIT> "(" (<DIGIT>|<HEXDIGIT>|structpart_or_varname()) ")" ";"
}

void set(): {}
{
    <KW_SET> "(" (<DIGIT>|<HEXDIGIT>)<KOMMA> arith_expression() ")" ";"
}

void get(): {}
{
    <KW_GET> "(" structpart_or_varname() <KOMMA> (<DIGIT>|<HEXDIGIT>) ")" ";"
}


void strcat(): {}
{
    <KW_STRCAT> "(" structpart_or_varname() (<KOMMA> (<STRING_LITERAL>|
              structpart_or_varname()))+ ")" ";"
}


void syscall():{}
{
    <KW_SYSCALL> "(" (<DIGIT>|<HEXDIGIT>) <KOMMA> structpart_or_varname() ")" ";"
}

// auslagern identifier in subfunctionname nicht nötig, da nur hier benutzt
void subfunctioncall(): {}
{
    <IDENTIFIER> "("  [ ( arith_expression()|<STRING_LITERAL>) ("," (arith_expression()|<
              STRING_LITERAL>))* ]  ")" ";"
}


void ifloop(): {}
{
    <KW_IF> "(" compare_expression() ")" "{"
    statements()
    "}" [  <KW_ELSE> "{" statements() "}"]
}

void whileloop(): {}
{
    <KW_WHILE>  "(" compare_expression() ")" "{"
    statements()
    "}"
}

void repeatloop(): {}
{
    <KW_REPEAT> "(" arith_expression() ")" "{"
    statements()
    "}"
}

void switchloop(): {}
{
    <KW_SWITCH> "(" arith_expression() ")" "{"
    ( <KW_CASE> <DIGIT> ":" ( statement() | <KW_BREAK> ";")*)*
    <KW_DEFAULT> ":" (statement()|<KW_BREAK>)*
    "}"
}
```

```
void doloop(): {}
{
    <KW_DO> "{"     statements() "}"
    <KW_WHILE> "(" compare_expression() ")"

}

void variable_type(): {}
{
    (<KW_INT> | <KW_STRING> | <KW_DWORD>| <KW_SWORD> | <KW_SBYTE> |<KW_DBYTE>|<KW_BYTE>)
}

void compare_expression() : {}
{
    arith_expression()
    [ (<OP_EQ>|<OP_GT>|<OP_LT>|<OP_GTEQ>|<OP_LTEQ>|<OP_NEQ>)
    arith_expression() ]
    [ (<OP_AND>|<OP_OR>) compare_expression() ]
}

void var_incr_decr(): {}
{
    structpart_or_varname() (<OP_INCR>|<OP_DECR>) ";"

}

void assignment(): {}
{
    structpart_or_varname() (<AS_NORM>|<AS_ADD>|<AS_SUB>|<AS_MUL>|<AS_DIV>) (
              arith_expression()|<STRING_LITERAL>) ";"
}

void structpart(): {}
{
    varname() ("." <IDENTIFIER>)+
}

// arithmetische Ausdrücke
void arith_function(): {}
{
    <OP_ABS>|<OP_SIGN>
}

void arith_expression(): {}
{

     arith_term() ((<OP_ADD>|<OP_SUB>)  arith_term())*
}

void arith_sign(): {}
{
    (<OP_ADD>|<OP_SUB>)
}

void arith_term(): {}
{
    arith_factor() ((<OP_MUL>|<OP_DIV>|<OP_MOD>) arith_factor())*
}


void arith_factor(): {}
```

```
{
     [ arith_function() ] "(" arith_expression() ")" | [arith_sign()]  (
               structpart_or_varname() [<OP_INCR>|<OP_DECR>] | <DIGIT> | <HEXDIGIT>)

}

// unterscheidung zwischen definition und deklaration anhand von LOOKAHEAD,
// first(2) sind identisch ist aber besser lesbar so
void struct_declaration_or_definition(): {}
{
    (LOOKAHEAD(3) struct_definition() | struct_declaration() ) ";"
}

// nicht struct_name benutzen, structurname wird erst hier definiert
void struct_definition(): {}
{
    <KW_STRUCT> <IDENTIFIER> "{" ((variable_type() | <KW_STRUCT> <IDENTIFIER> ) <IDENTIFIER
               > ";")+ "}"
}

// hier ist die structur bereits definiert
void struct_declaration(): {}
{
     <KW_STRUCT> struct_name() varname() (<KOMMA> varname())*
}
```

A.3. precompiler.jtb

Der Quellcode der Präprozessor-Grammatik für JavaCC/JTB.

```
options {
  LOOKAHEAD = 1;
  CHOICE_AMBIGUITY_CHECK = 2;
  OTHER_AMBIGUITY_CHECK = 1;
  STATIC = false;
  DEBUG_PARSER = false;
  DEBUG_LOOKAHEAD = false;
  DEBUG_TOKEN_MANAGER = false;
  ERROR_REPORTING = true;
  JAVA_UNICODE_ESCAPE = false;
  UNICODE_INPUT = false;
  IGNORE_CASE = false;
  USER_TOKEN_MANAGER = false;
  USER_CHAR_STREAM = false;
  TOKEN_MANAGER_USES_PARSER = true;
  BUILD_PARSER = true;
  BUILD_TOKEN_MANAGER = true;
  SANITY_CHECK = true;
  FORCE_LA_CHECK = false;
}

PARSER_BEGIN(NnqcPreProcessor)
package preprocessor;


public class NnqcPreProcessor  {

}
```

```
PARSER_END(NnqcPreProcessor)


// tokens for every state

// comments
<*> TOKEN : /*comments*/
{
    <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
    |   <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/" >
}


// default tokens

<DEFAULT> TOKEN: { <PRE_BEGIN : "#" > : PREPROCESSOR }
<DEFAULT> TOKEN: { <REST: ~["#"]> }

// preprocessor tokens
<PREPROCESSOR> TOKEN: { <INCLUDE: "include"> :PREINCLUDE }
<PREPROCESSOR> TOKEN: { <DEFINE: "define">: PREDEFINE}
<PREPROCESSOR> SKIP: {" "|"\t"}

// preinclude tokens
<PREINCLUDE> TOKEN: { <FILENAME: ([ "a" - "z", "0" - "9" , "_", "-", "A"- "Z", "/", ".", "
          \\"])+ > }
<PREINCLUDE> TOKEN: { <QUOTES: "\""> }
<PREINCLUDE> SKIP: { <END_PREIN: "\n" |"\r\n" >: DEFAULT }
<PREINCLUDE> SKIP: {" "|"\t"}

// predefine tokens
<PREDEFINE> TOKEN: {<DIGIT: (["0" - "9"])+ > }
<PREDEFINE> TOKEN: {<IDENTIFIER: (["a"-"z","A"-"Z","_","-"])(["a"-"z","A"-"Z","_","-"]|<
          DIGIT>)*> }
<PREDEFINE> TOKEN: {<RBRACKETOPEN: "(">}
<PREDEFINE> TOKEN: {<RBRACKETCLOSE: ")">: PREDEFINEFUNCTION }

<PREDEFINE> TOKEN: { <KOMMA: ","> }
<PREDEFINE> TOKEN: {<STRING_LITERAL: "\"" ( ~["\"","\\","\n","\r"] | "\\" ( ["n","t","b","r
          ","f","\\","\'","\""] | ["0"-"7"] (["0"-"7"])?  | ["0"-"3"] ["0"-"7"] ["0"-"7"]
          |( ["\n","\r"] | "\r\n")))* "\""> }
<PREDEFINE> SKIP: { <END_PREDEF: "\n" |"\r\n" >: DEFAULT }
<PREDEFINE> SKIP: {" "|"\t"}

<PREDEFINEFUNCTION> TOKEN: { <FUNCTIONBODY: (~["\n","\r"])+> }
<PREDEFINEFUNCTION> SKIP: { <END_PREDEFUNC: "\n" |"\r\n" >: DEFAULT }

void Input(): {}
{
(pre_statement()| codeblock()|comment())* <EOF>
}

void comment(): {}
{
    <SINGLE_LINE_COMMENT>|<MULTI_LINE_COMMENT>
}

void pre_statement(): {}
{
    <PRE_BEGIN> (include()|LOOKAHEAD(3) define_function()| LOOKAHEAD(3) define_literal())
}

void include(): {}
```

```
{
    <INCLUDE> <QUOTES> <FILENAME> <QUOTES>
}

void define_literal(): {}
{
    <DEFINE> <IDENTIFIER> (<DIGIT> |<STRING_LITERAL>|<IDENTIFIER>)

}

void define_function(): {}
{
    <DEFINE> <IDENTIFIER> <RBRACKETOPEN> [ <IDENTIFIER> ( <KOMMA> <IDENTIFIER>)*]
    <RBRACKETCLOSE>  <FUNCTIONBODY>
}
void codeblock(): {}
{
 (LOOKAHEAD(2)<REST>)+
}
```

A.4. Pseudocode für die Zielcodeerzeugung

A.4.1. Do-while und while-do-Schleifen

```
dowhiledo(compare_expression, Statements)
  cp = getFreeStackplace
  n = getNextLoopNum(Contants.KW_WHILE)
  code("WHILEnSTART:")
  do_compare_expression(compare_expression, cp)
  code("brtst LTEQ, RLOOPnEND, _astack[cp]")
  do_statements(Statements)
  code("jmp RLOOPnStart")
  code("WHILEnEnd:")
  freeStackplace(cp)

dodowhile(compare_expression, Statements)
  cp = getFreeStackplace
  n = getNextLoopNum(Contants.KW_WHILE)
  code("DOnSTART:")
  do_statements(Statements)
  do_compare_expression(compare_expression, cp)
  code("brtst GT, DOnSTART, _astack[cp]")
```

A.4.2. Zuweisungen

```
doAssignment(Var/Struct, OP, assvalue)
  if assvalue.type = string_literal
  code("mov `` + getRealname(Var/Structpart) + ``, assvalue")
  else if assvalue.type = arith_expression
   cp = getFreeStackplace
   do_arith(assvalue,cp)
   switch(OP.type)
        case ``+=''
            code("add`` + ``_astack[cp]'' + ``, _astack[cp]" +
                        getRealname(Var/Structpart))
            break;
        case ``-=''
            code("sub`` + ``_astack[cp]'' + ``, _astack[cp]" +
                        getRealname(Var/Structpart))
            break;
        case ``*=''
```

```
            code("mul`` + ``_astack[cp]'' + ``, _astack[cp]" +
                        getRealname(Var/Structpart))
            break;
        case ``/='':
            code("div`` + ``_astack[cp]'' + ``, _astack[cp]" +
                        getRealname(Var/Structpart))
            break;
        case ``=':
            break;

   code("mov`` + getRealname(Var/Structpart) + ``, _astack[cp]")

  else
   error
```

A.4.3. If-then-else

```
do_ifthenelse(compare_expression, IfStatements, ElseStatements)
  cp = getFreeStackplace
  n = getNextLoopNum(Contants.KW_IF)
  do_compare_expression(compare_expression, cp)
  code("brtst GT, IFnSTART, _astack[cp]")
  code("jmp IFnELSE:")
  code("IFnSTART:")
  do_statements(IfStatements)
  code("jmp IFnEND")
  code("IFnELSE:")
  do_statements(ElseStatements)
  code("IFnEND:")
  freeStackplace(cp)
```

A.4.4. switch-case

```
do_switch(arith_expression, defaultStatements, value0, Statement0,
             value1, Statement1, ....)
  cp = getFreeStackplace
  n = getNextLoopNum(Contants.KW_IF)
  int count = 0;
  do_arith_expression(arith_expression, cp)
  code("brcmp EQ, SWITCHn_count++, value1, _astack[cp]")
  code("brcmp EQ, SWITCHn_count++, value2, _astack[cp]")
  code("brcmp EQ, SWITCHn_count++, value3, _astack[cp]")
```

```
doStatements(defaultStatements)
code("jmp SWITCHnEND")
code("SWITCHn_count0:")
do_Statements(Statement0)
code("SWITCHn_count0:")
do_Statements(Statement0)
....
if lastStatement == break
code("jmp SWITCHnEND")
code("SWITCHn_1:")
do_Statements(Statement1)
if lastStatement == break
code("jmp SWITCHnEND")
\ldots
code("SWITCHnEND:")
freeStackplace(cp)
```

B. NNQC-Beispiele

B.1. Helloworld

Dieses Programm gibt für eine Sekunde den String ”hallo welt” auf der Anzeige des NXT aus.

```
define SYSCALL_DRAW_TEXT       13
struct TLocation {
        sword   x;
        sword   y;
};

struct TDrawText {
        byte    result;
        struct TLocation location;
        string  text;
        dword   options;
};

struct TDrawText dtargs;

task main() {
        dtargs.location.x = 0;
        dtargs.location.y = 0;
        dtargs.text = "hallo welt";
        syscall(SYSCALL_DRAW_TEXT,dtargs);
        wait(1000);
}
```

B.2. Playtone

Dieses Testprogramm spielt für eine Sekunde einen Ton.

```
#define SoundPlayTone 10
struct SoundPT {
        sbyte   result;
        sword   freq;
        sword   time;
        byte    loop;
        byte    vol;
};

struct SoundPT sound;

task main() {
        sound.freq=440;
        sound.time=1000;
        sound.loop=0;
        sound.vol=3;
        syscall(SoundPlayTone,sound);
        wait(1000);
}
```

Glossar

I^2C Inter-Integrated Circuit. Ein serielles Bussystem geringer Geschwindigkeit, oft benutzt zum Anschluss von Sensoren.

Abstrakter Syntaxbaum(AST) (Zwischen-) Darstellung eines Quellcodes in Form eines Baumes. Dabei fehlen – im Gegensatz zum konkreten Syntaxbaum – oft schon Details wie Klammern, Kommata oder Semikolon, die inhaltlich nicht wichtig sind.

API Application Programming Interface, eine Schnittstelle auf Quelltexteben zur Anwendungs-Programmierung.

Attributierte Grammatik Eine kontextfreie Grammatik die um zusätzliche semantische Regeln erweitert wurde. Dies dient dazu Regeln zu überprüfen, die mit einer kontextfreien Grammatik nicht formuliert werden können (z.B. Typüberprüfungen während Zuweisungen).

BNF Backus-Naur-Form: Eine Metasprache zur formalen Definition von kontextfreien Programmiersprachen

Bottom-Up-Analyse Konstruktion des Parse-Baumes von den Blättern her. Dazu werden LR- bzw. LALR-Grammatiken benutzt.

Bumper Ein Bumper ist ein einfacher Schalter, der verwendet wird, um Kollisionen mit einem Hindernis zu erkennen. Der Schalter wird in diesem Falle (konstruktionsbedingt) geschlossen.

EBNF Erweiterte Backus-Naur-Form: Ein erweiterte Form der BNF. Hauptsächliche Neuerungen sind Möglichkeiten der Darstellung von optionalen und iterativen Konstrukten.

Error Recovery Hiermit wird das Verhalten eines Compilers bezeichnet, nach einem Fehler beim Parsen des Quellcodes an einer späteren Stelle das Parsen fortzuführen. Dadurch lassen sich in einem Compilerlauf mehrere Fehler feststellen (und nicht nur der erste gefundene).

JRE Java Runtime Environment. Hauptbestandteile sind die Java Virtual Machine und die Java API.

Konkreter Syntaxbaum Die komplette Ableitung in Form eines Baumes.

LALR(1)-Grammatik LR(1) Grammatik mit Lookahead.

LL(k)-Grammatik L steht für von-links-nach-rechts, L für Linksableitung

LR(1)-Grammatik L steht für von-links-nach-rechts, R für Rechtsableitung in Umgekehrter Reihenfolge.

Metacompiler Ein Compiler, der andere Compiler erzeugt.

NBC Next Byte Code. Eine Assembler nachempfundene Programmiersprache für den LEGO Mindstorms NXT.

NNQC Nearly Not Quite C. Eine NQC nachempfundene Programmiersprache für den LEGO Mindstorms NXT.

Node Node ist das Interface das von Klassen, die Knoten in einem Syntaxbaum bei JTB darstellen, implementiert werden muss. In diesem Zusammenhang kann es synonym mit dem Begriff Knoten benutzt werden.

NodeToken NodeToken ist die Klasse die in einem von JTB erzeugten Baum genutzt wird, um die Token – also die erkannten Strings – zu speichern. Der gesamte Quelltext liegt in den Blättern des Baumes in Form von NodeTokens vor.

Non-Terminal Anderes Wort für Produktion einer formalen Grammatik.

NQC Not Quite C. Eine C nachempfundene Programmiersprache für den LEGO Mindstorms RCX.

Panikmodus Der Panikmodus ist eine Strategie für die Fehlerbehandlung in Parsern. Beim Panikmodus wird ein Teil der Eingabe überlesen, bis ein sogenannter Synchronisationspunkt gefunden wurde. Das ist meist ein Semikolon oder eine schließende geschweifte Klammer, also Zeichen, deren Bedeutung innerhalb des Quellprogrammes der Abschluss eines bestehenden Konstruktes ist.

Parsebaum Anderes Wort für Syntaxbaum.

Produktionen Bei einer Produktion wird in einer formalen Grammatik eine Menge von Zeichen durch eine andere ersetzt.

Terminal Ein Zeichen bzw. eine Zeichenkette, die nicht weiter reduziert werden kann.

Token Ein Token ist eine Zeichenkette, die innerhalb der Grammatik eine Bedeutung hat.

Top-Down-Analyse Konstruktion des Parse Baumes vom Knoten her. LL(k)-Grammatiken kommen hierfür zum Einsatz.

UPN Umgedrehte Polnische Notation ist eine mathematische Notation, bei der die Therme in Postfix-Schreibweise (Also die Operatoren nach den Operanden) vorliegen.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 24. August 2006	
Ort, Datum	Unterschrift