# Projektarbeit Compilerbau

Jonas Wenner, Jens Marvin Reuter, Noah Hoffmann

8. Oktober 2018

# Inhaltsverzeichnis

**Zusammenfassung**

Als Abschlussprojekt des Moduls 'Compilerbau' unter Leitung von Thorsten Jakobs, M.Sc., htw saar wurde dieser Compiler zur von uns eigens entwickelten Sprache *UCBS* (**U**nsere **C**ompiler**b**au-**S**prache) entwickelt, mit den in *1. Anforderungen* beschriebenen Voraussetzungen und Zielen. Die Sprache besitzt eine C-ähnliche Syntax, wurde jedoch auf wesentliche Elemente wie Grundrechenarten, Funktionen, if-Statements und eine Form der Schleife reduziert. Weiterhin vorhanden sind Bezeichner, Konstanten und Variablen. Die Klammerung wird ebenfalls berücksichtigt.

Diese Dokumentation behandelt die Anforderungen an den Compiler, den theoretischen Hintergrund, die Konzeption der Entwicklung, unsere verwendeten Technologien, sowie unsere Vorgehensweise während der Entwicklung und die Probleme, auf die wir gestoßen sind und unsere Lösungsansätze. Ausgangs wird auch die Handhabung des Compilers näher erläutert.

# 1 Anforderungen

Um zu gewährleisten, dass alle Modulteilnehmer über die gleichen Voraussetzungen verfügen und nach festgelegten Kriterien bewertet werden können, wurden durch den Modulverantwortlichen einige Rahmenbedingungen vorgegeben. Die Prüfleistung besteht dabei aus der Entwicklung einer einfachen Programmiersprache, dem Anfertigen einer Dokumentation sowie der Präsentation der Ergebnisse. Die Anforderungen an das Projekt werden in den folgenden Abschnitten aufgeführt und erläutert.

## 1.1 Infrastruktur

Das Ziel ist die Entwicklung eines Compiler-Frontends, welches Quelltext in der von uns entwickelten Programmiersprache zu Jasmin-Code übersetzt. Jasmin ist ein Assembler für die Java Virtual Machine, der Quelltext in einer Assembler-ähnlichen Syntax zu Bytecode übersetzt. Dieser Bytecode kann schließlich von der Java Virtual Machine ausgeführt werden.

## 1.2 Programmiersprache

Die von uns entwickelte Programmiersprache besitzt folgende Komponenten:

- Bezeichner
- Konstanten
- Variablen
- Arithmetik mit Grundrechenarten
- Klammerung
- Funktionen
- if-else-Anweisung
- while-Schleife

Die Entwicklung wird mit 50% in der Gesamtbewertung gewichtet. In einem späteren Kapitel dieser Dokumentation wird erläutert, wie die Programmiersprache verwendet wird.

## 1.3 Dokumentation

Es ist eine Dokumentation anzufertigen, die Aufschluss über die Konzeption der Entwicklung, die verwendeten Technologien, das Vorgehen während der Entwicklung, aufgetretene Probleme und Lösungsansätze zu diesen sowie die Verwendung des Compilers gibt.
Die Dokumentation wird mit 30% in der Gesamtbewertung gewichtet.

## 1.4 Präsentation

In einer circa 20-minütigen Präsentation werden die verwendeten Technologien, wichtige Entwicklungsschritte, sowie aufgetretene Probleme erläutert. Außerdem wird der Compiler vorgeführt.
Die Präsentation wird mit 20% in der Gesamtbewertung gewichtet.

# 2 Theoretischer Hintergrund

Um die nachfolgende Abschnitte verständlich zu machen ist eine kurze Erläuterung des theoretischen Hintergrunds sinnvoll.

## 2.1 Verallgemeinerung

Der zu entwickelnde Compiler nimmt als Eingabe eine Text-Datei mit Instruktionen in der Ausgangssprache. Der Compiler verarbeitet diese Eingabe und gibt als Ausgabe eine Datei mit Anweisungen in der Zielsprache zurück. Diese Verarbeitung kann grob in drei Abschnitte eingeteilt werden.

### 2.1.1 Lexikalische Analyse

Der erste Schritt ist dabei die sogenannte lexikalische Analyse. Die Eingabedatei ist für den Computer zunächst nur eine Kette von Zeichen. Diese Zeichen sind in sogenannte Tokens aufzuteilen. Beispielhaft könnte die Eingabe

```
a = b + c ;
```

zu den folgenden Tokens aufgelöst werden.

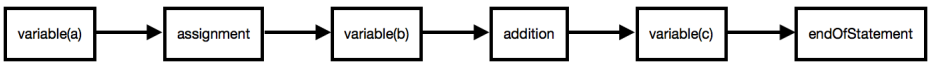


Abbildung 1: Kurzes Beispiel für lexikalische Analyse

Der Teil des Compilers, der diese Aufgabe übernimmt, wird *Lexer* genannt. Der Lexer überprüft für jedes Zeichen der Eingabedatei, welchem Token dieses zugeordnet werden kann. Dies geschieht auf Grundlage von definierten Regeln, welche mit sogenannten *regulären Ausdrücken* formuliert werden. Ein regulärer Ausdruck besteht aus einer Sequenz von Zeichen, die ein bestimmtes Muster definieren. Reguläre Ausdrücke können auch rekursiv verwendet werden, um auf Grundlage einfacher regulärer Ausdrücke komplexere zu bilden. Beispiele für reguläre Ausdrücke werden im Kapitel *5.1. Erste Schritte: Auswertung von arithmetischen Ausdrücken* aufgeführt.

### 2.1.2 Syntaktische Analyse

Im zweiten Schritt der Verarbeitung wird die *syntaktische Analyse* durchgeführt. Dabei wird überprüft, ob die Eingabedatei einer definierten Grammatik entspricht, während bei der lexikalischen Analyse lediglich die übergebene

Zeichenfolge in Tokens aufgeteilt wird, nicht jedoch überprüft wird, ob diese der Grammatik entsprechend zusammengeführt werden können.
Bei einer erfolgreichen Auswertung wird die Folge an Tokens, die bei der lexikalischen Analyse ermittelt wurde, in einen Syntaxbaum übergeführt. Dies ist notwendig, um die Tokens in der richtigen Reihenfolge auszuwerten.

### 2.1.3 Syntaxgesteuerte Übersetzung

Die eigentlich Erzeugung von Code in der Ausgangssprache findet zuletzt statt. Hierbei wird jeder Knoten des Syntaxbaum betrachtet und je nach Kontext eine Zeichenkette mit Anweisungen in der Zielsprache generiert. In diesem Schritt findet ebenfalls die semantische Analyse statt, beispielsweise die Überprüfung, ob eine aufgerufene Variable vorher deklariert wurde.

# 3 Konzeption der Entwicklung

## 3.1 Allgemeiner Grundsatz

Während der Entwicklung versuchten wir uns möglichst kleine Zwischenziele zu setzen, um die Aufgabe in überschaubare, einfach zu lösende Teilprobleme zu unterteilen. Rückblickend war dies eine sinnvolle Strategie, die zum gewünschten Ziel führte.

## 3.2 Wahl der Entwicklungswerkzeuge

Bezüglich der Entwicklungswerkzeuge wurden uns verschiedene Möglichkeiten vorgestellt. Die erste Möglichkeit bestand darin das Tool ”lex” beziehungsweise dessen Open-Source-Implementierung ”flex” als Lexer-Generator und das Tool ”yacc” beziehungsweise dessen Open-Source-Implementierung ”bison” als Parser-Generator zu nutzen. Die zweite Möglichkeit bestand in der Nutzung des Werkzeuges ANTLR, das die Funktionalitäten der zuvor genannten Werkzeuge in einem Programm zusammenfasst.
Da ANTLR automatisiert auf Grundlage einer Datei, die die von uns formulierte Grammatik der Sprache enthält, Lexer sowie Parser zeitgleich erstellen kann, entschieden wir uns für diese Variante. Der Vorteil besteht dabei darin, dass zwei essentielle Teile des Compilers von einem Tool übernommen werden.

## 3.3 Herkunft des Namens UnsereCompilerbauSprache

Für die Programmiersprache wurde ursprünglich der Name C= gewählt (gesprochen: C equal). In einem vorherigen Schritt sollte die Programmiersprache C - - genannt werden, in Anlehnung an den verkleinerten Sprachumfang der Sprache C sowie der Namensgebung von C++. Da C - - jedoch bereits als Zwischensprache existiert, wurde die im Rahmen des Moduls Compilerbau entwickelte Sprache in Anlehnung an C# (# ist auch als vier aneinandergereihte Additionssymbole zu interpretieren) zunächst C= genannt, wobei das = vier aneinandergereihte Subtraktionszeichen darstellen soll. Diese Idee wurde jedoch im Laufe der Projektarbeit verworfen und für den Namen der Programmiersprache wurde sich auf *UnsereCompilerbauSprache* geeinigt, um einen etwas besser zum Projekt passenden Namen zu verwenden

## 3.4 Sprachdesign

Es wurde sich darauf geeinigt, dass die Syntax der von uns entworfenen Ausgangssprache der Syntax der Programmiersprache C ähnlich sein soll, da dies gleich mehrere Vorteile hervorbringt:

- Alle Projektteilnehmer sind mit der Sprache C vertraut
- C ist eine der am weitesten verbreiteten compilierten Sprachen
- Die Sprache C besitzt aufgrund seiner Klammerregeln eine gute Lesbarkeit

## 3.5 Implementierung

Unser Compiler verwendet im Zusammenspiel mit dem von ANTLR generierten Lexer und Parser einen Visitor für die syntaxgesteuerte Übersetzung. Die Alternative dazu besteht in einem Listener, der sich von einem Visitor dahingehend unterscheidet, dass die verschiedenen Methoden die auf die Knoten des Syntaxbaumes angewandt werden, keinen Rückgabewert besitzen und die Ergebnisse dieser Methoden somit separat gespeichert werden müssen. Der Vorteil eines Listeners hingegen besteht darin, dass untergeordnete Knoten im Syntaxbaum automatisch bearbeitet werden und nicht wie beim Visitor explizit besucht werden müssen. Nach der Abwägung dieser Vor- und Nachteile entschieden wir uns für einen Visitor, da dieser uns als die einfachere Methode erschien.

# 4 Verwendete Technologien

## 4.1 Eclipse

Zur Entwicklung wurde die integrierte Entwicklungsumgebung Eclipse in der aktuellen Version (4.8.0) genutzt. Eclipse bietet mehrere Vorteile, die ein einfacheres und effizienteres Entwickeln erlauben. Eclipse enthält einen Datei-Explorer, mit dessen Hilfe die Navigation durch die verschiedenen Verzeichnisse zusammen mit Editor und Konsole innerhalb eines Fensters geschehen kann. Des Weiteren bietet Eclipse automatische Vorschläge zur Code- und Textvervollständigung an, was die Schreibgeschwindigkeit und damit die Arbeitseffizienz erhöht.
Weiterhin kann Eclipse durch Plug-Ins erweitert werden. Als Plug-In nutzen wir beispielsweise das Test-Framework TestNG, um verschiedene Testfälle automatisiert zu prüfen.

Ein großer Nachteil von Eclipse ist die von uns erfahrene Unzuverlässigkeit. Die IDE wirft immer wieder nicht nachvollziehbare, manchmal sogar gar keine Fehlermeldungen, wenn Aufgaben wie das Öffnen und Kompilieren von Projekten fehlschlagen. Des weiteren trat unabhängig voneinander auf verschiedenen Rechnern mit verschiedenen Betriebssystemen (getestet u.A. unter Ubuntu Linux, Arch Linux und Mac OS X) das Problem auf, dass das Projekt nicht geöffnet werden konnte. Besonders seltsam war jedoch die Situation, dass sich das Projekt mit der *'Eclipse IDE for C/C++ Developers'* öffnen ließ, nicht jedoch mit *'Eclipse IDE for Java Developers'*. Letztendlich war die Arbeit mit Eclipse unter Arch Linux nicht möglich, sodass andere Projektteilnehmer die Aufgabe des Kompilierens übernehmen mussten.

## 4.2 TestNG

TestNG ist ein Framework, das dem zu testenden Programm Eingabewerte übergibt und die tatsächlichen Ergebnisse des Programms mit den erwarteten Ergebnissen abgleicht. Dabei können beliebig viele Testszenarien geprüft werden. Es werden sogenannte positive Tests, also solche, bei denen die Eingabe ein erwartetes Ergebnis hervorruft, als auch negative Tests, bei denen geprüft wird, ob nicht vorhergesehene Eingaben mit entsprechenden Fehlermeldungen korrekt behandelt werden, durchgeführt. TestNG wurde von uns in Version 6.14 genutzt.
Als Beispiel: Der Quelltext

```
println(1+4);
```

sollte als Ergebnis ”5” auf der Konsole ausgeben. TestNG übergibt den Quelltext an unseren Compiler, ruft Jasmin auf, führt das Programm aus und gleicht dann die Ergebnisse ab. Wenn das Ergebnis ”5” ist, gilt der Test als bestanden, andernfalls als durchgefallen.

## 4.3 ANTLR

ANTLR steht für *ANother Tool for Language Recognition* und ist ein Lexer- und Parsergenerator. ANTLR generiert auf Grundlage einer Grammatik-Datei Java-Code, der einen entsprechenden Lexer sowie ein Template für Teile des Parsers implementiert. Dadurch, dass die von ANTLR generierten Programme aus einer Eingabedatei in der Ausgangssprache einen Syntaxbaum erstellen können, besteht der größte Arbeitsaufwand daraus, Grammatiken für ANTLR zu formulieren und Teile des Parsers (in unserem Fall ein Visitor) zu implementieren. Es wurde die ANTLR-Version 4.7.1 verwendet.

## 4.4 Jasmin

Jasmin ist ein Assembler für die Java Virtual Machine. Dabei werden Textdateien mit Anweisungen in Jasmin-Syntax zu Bytecode übersetzt. Dieser Bytecode kann von der JVM ausgeführt werden.

## 4.5 Java

Die Programmiersprache Java wurde für das Projekt gewählt, da der von ANTLR generierte Code ebenfalls in Java erstellt wird. Es ist teilweise notwendig von Klassen des von ANTLR generierten Code abzuleiten, weshalb die Wahl einer anderen Programmiersprache als Java Problemstellungen wie die Code-Kompatibilität und Portabilität dargestellt hätte. Des Weiteren muss die Java Runtime Environment ohnehin genutzt werden, um ANTLR, Jasmin sowie schließlich die Kompilate des Compilers auszuführen.

## 4.6 Java Virtual Machine

Die Java VM ist ein Zwischenschritt beim Ausführen von Java-Code. Eine Java-Quelldatei (.java) wird zunächst durch den Java-Compiler zu Bytecode (.class) übersetzt und dann von der Java VM interpretiert. Dieser Zwischenschritt ermöglicht eine Plattformunabhängigkeit, da die Kompilate (.class-Dateien) nicht maschinenspezifisch übersetzt werden. Plattformabhängiger Code wird erst von der Java VM ausgeführt, wenn diese den Bytecode interpretiert, weshalb der Bytecode für jeden Computer maschinenunanbhängig

generiert und ausgeführt werden kann, solange für die entsprechende Maschine die Java VM verfügbar ist.
In diesem Projekt wird die Java VM verwendet, um die Kompilate unseres eigenen Compilers, nachdem diese von Jasmin bearbeitet wurden, auszuführen.
Zu beachten ist, dass die Java VM stack-basiert operiert und nicht wie der x86-Befehlssatz mit Registern arbeitet. Dies ist bei der Implementierung des Compilers zu beachten und erfordert eine besondere Denkweise und Kenntnis der Stack-Struktur.

## 4.7 Git

Nach einiger Zeit der Arbeit am Projekt wurde klar, dass es auf Dauer sehr ineffizient ist, die Aktualisierungen über eine WhatsApp-Gruppe zu verteilen. Daher haben wir über die Plattform *GitHub* ein Repository erstellt, in dem alle Änderungen zentral verwaltet und verteilt werden.

# 5 Vorgehen während der Entwicklung und Implementierung der Sprache

Dieser Abschnitt beschreibt unser Vorgehen während der Entwicklung der Programmiersprache. Ein Unterabschnitt stellt dabei jeweils einen Entwicklungsschritt des Compilers da. Um diese Abschnitte besser verständlich zu machen, wird teilweise von Kapitel 7. *Verwendung der UnsereCompilerbau-Sprache* vorgegriffen.

## 5.1 Erste Schritte: Auswertung von arithmetischen Ausdrücken

Der Vorgehensweise unter *3.1. Allgemeiner Grundsatz* entsprechend, implementierten wir zuerst die unserer Sicht nach grundlegendste Fähigkeit einer Programmiersprache: das Auswerten von arithmetischen Ausdrücken. Es soll möglich sein, Ausdrücke, die die Grundrechenarten sowie eine beliebig tiefe Schachtelung von Klammern enthalten, zu erkennen und der Operatorenpriorität entsprechend die gelesenen Tokens in einem Syntaxbaum zusammenzuführen. Dazu formulierten wir folgende Grammatik:

```
//Datei: Arithmetic.g4
grammar Arithmetic;

////////////////////////////////////////////////////
// beliebige Folge der Ziffern 0 bis 9
INTEGER : [0-9]+ ;
// ueberspringt Leerzeichen, Tabstops sowie Linefeeds
WS : [ \t\r\n]+ -> skip ;
// oeffndende runde Klammer
LPAREN : '(';
// schliessende runde Klammer
RPAREN : ')';
////////////////////////////////////////////////////
//mathematische Operatoren
PLUSOP : '+';
MINOP : '-';
MULTOP : '*';
DIVOP : '/';
////////////////////////////////////////////////////
//Regeln fuer math. Ausdruecke
expression: INTEGER
        | LPAREN expression RPAREN
        | expression DIVOP  expression
        | expression MULTOP expression
        | expression MINOP  expression
        | expression PLUSOP expression
        ;
////////////////////////////////////////////////////
```

Eine Grammatik für ANTLR hat folgenden Aufbau:

Die Definition der Grammatik beginnt mit dem Schlüsselwort ”grammar” sowie dem Namen der Grammatik. Dabei gilt es zu beachten, dass die Datei den gleichen Namen wie die Grammatik selbst sowie die Dateiendung ”.g4” besitzt. Diese Deklaration wird mit einem Semikolon abgeschlossen.
Zeilen, die mit ”//” beginnen, sind Kommentare und haben keinen Einfluss auf die Grammatik. Sie dienen lediglich als Erläuterungen und zur Formatierung, um die Lesbarkeit zu verbessern.
Auf die Deklaration dürfen beliebig viele Regeln für die Grammatik folgen. Zur Formulierung der Regeln werden reguläre Ausdrücke genutzt. Beispielsweise besagt die sechste Zeile, dass es eine Regel INTEGER gibt, wobei ein INTEGER sich aus einer beliebigen Folge der Ziffern von 0 bis 9 zusammensetzt. Der reguläre Ausdruck [0-9] gibt an, dass ein beliebiges Zeichen im Bereich von 0 bis 9 vorkommen darf. Das abschließende ”+” bedeutet, dass es sich um eine Kette dieser Zeichen, die beliebig lange ist, jedoch mindestens die Länge 1 besitzt, handelt.
Die Regel WS (Whitespace) besagt, dass bestimmte Zeichen, die nur der Formatierung dienen, ignoriert werden, da sie für das Übersetzen einer Quelldatei keine Bedeutung haben.
Die darauffolgenden Regeln sind Aliase für die Zeichen, die in arithmetischen Ausdrücken verwendet werden. Diese Auslagerung steigert unseres Erachtens nach die Lesbarkeit der letzten Regeln dieses Beispiels, sind aber nicht zwingend notwendig.
Die wohl relevanteste Regel trägt den Namen ”expression” und legt fest, wie sich ein arithmetischer Ausdruck zusammensetzen kann. Im Vergleich zu den vorherigen Regeln, wurde hier von der Möglichkeit, mehrere alternative Möglichkeiten anzugeben, Gebrauch gemacht. Die Regel besagt, dass ein arithmetischer Ausdruck entweder aus

- einer Zahl,
- einem geklammerten Ausdruck,
- einer Division mit zwei Operanden,
- einer Multiplikation mit zwei Operanden,
- einer Subtraktion mit zwei Operanden
- oder einer Addition mit zwei Operanden

besteht. Durch die Rekursion (die Regel verweist auf sich selbst) ist eine beliebige Länge des Ausdruckes möglich.

Eine Besonderheit, die zu beachten ist, besteht in der Rangfolge der Operatoren. Die höchste Bindung besitzt ein geklammerter Term, die nächst höchste Bindung Divisionen und Multiplikationen. Die schwächste Bindung besitzen Subtraktionen und Additionen.
Damit diese Priorität gewährleistet werden kann, sind die Regeln in dieser bestimmten Reihenfolge notiert. ANTLR versucht immer zuerst die "oberste" Regel anzuwenden, daraufhin die darunter stehende usw.. Deshalb wird der folgende Ausdruck

```
2*10-48*(4-1)-16/4
```

zu diesem Syntaxbaum aufgelöst:

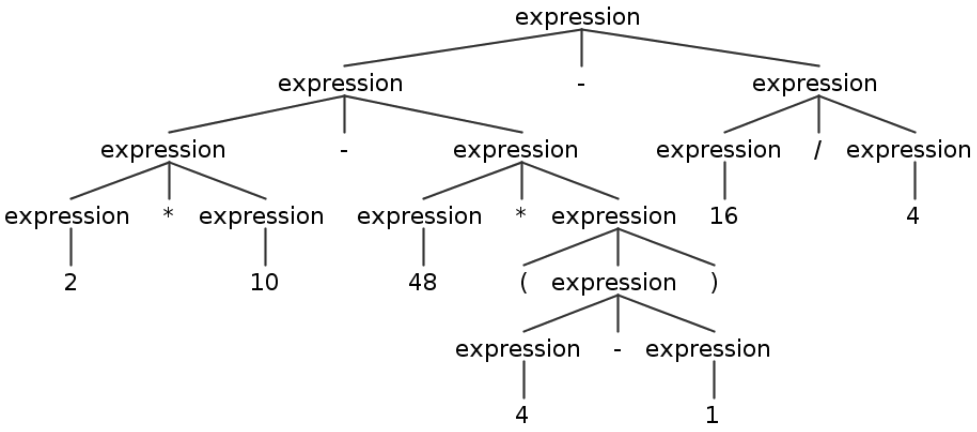


Abbildung 2: Syntaxbaum für beispielhaften arithmetischen Ausdruck

Des Weiteren ist zu beachten, dass arithmetischen Ausdrücke gleicher Operatorenpriorität von links nach rechts ausgewertet werden müssen. Als Beispiel lässt sich hier der Term

```
8-5+2
```

nennen. Sowohl die Subtraktion als auch die Addition besitzen die gleiche Bindung. Wenn man hier fälschlicherweise zuerst die Addition, also 5+2, was 7 ergibt, auswertet und das Ergebnis dieser Operation von 8 subtrahiert, erhält man als Gesamtergebnis 1. In der korrekten Reihenfolge erfolgt zuerst die Subtraktion, also 8-5, was 3 ergibt, und erst darauf die Addition von 1, was als Gesamtergebnis 4 liefert. Um diese Fehlerquelle auszuschließen, sind in unserer Grammatik die Regel für Subtraktion vor der für Addition und die Regel für Division vor der für Multiplikation definiert. Da bei reinen Additionen bzw. reinen Multiplikationen die Auswertungsreihenfolge tatsächlich

keine Rolle spielt, eine Subtraktion jedoch vor einer Additionen (vgl. Beispiel oben) ausgewertet werden muss, sorgt die Reihenfolge der Regeln in der Grammatik für eine korrekte Auswertung.
Ein analoges Beispiel zu Divisionen und Multiplikationen ist

```
8/2*4
```

Auch hier ergibt sich ein ähnliches Problem wie beim vorherigen Beispiel. Wird zuerst die Multiplikation (2*4) ausgeführt und erst danach die Division (also 8/8 in diesem Fall), ist das Endergebnis 1 und nicht wie in der richtigen Reihenfolge 16.
Nach diesen Schritten sind wir also in der Lage einen Syntaxbaum zu erstellen. Der nötige Programmcode dafür wird von ANTLR automatisiert erstellt. Dabei ist die Eingabe dieses Programmes der auszuwertende Ausdruck. Die Ausgabe ist der Syntaxbaum, der zu Testzwecken auch als Grafik (vgl. Abbildung oben) ausgegeben werden kann.
Der nächste wichtige Schritt der Übersetzung besteht nun in der Auswertung dieses Baumes. Dazu wird der Baum als Datenstruktur betrachtet. ANTLR liefert dabei mehrere Methoden, die auf Instanzen der Klasse "ParseTree" angewandt werden können. Jedes Token, das in der Grammatik definiert wurde und durch einen Knoten im Baum repräsentiert wird, besitzt eine sogenannte Visit-Methode. Diese Methode gibt eine Kette mit Zeichen zurück, wobei diese Zeichenkette die Anweisungen in der Zielsprache (Jasmin) enthält. Das Abarbeiten dieser Visit-Methoden in der richtigen Reihenfolge bildet die Grundlage für die Übersetzung, da hiernach alle Instruktionen in der Zielsprache zusammengesetzt sind. Was genau in einer Visit-Methode passiert, wird vom Entwickler festgelegt. ANTLR stellt sogehesen nur eine Vorlage zur Verfügung.
Um diese korrekte Reihenfolge zu gewährleisten, muss eine Anfangsregel (Startaxiom) festgelegt sein. In diesem Beispiel wurde festgelegt, dass der Programmstart - sprich der Wurzelknoten des Baumes - eine "expression" sein muss. Das bedeutet, das zunächst die Visit-Methode des Wurzelknoten aufgerufen wird. Damit nun auch die inneren Knoten des Syntaxbaums berücksichtigt werden, ist der Aufruf einer weiteren von ANTLR generierten Methode notwendig. Jeder Knoten des Baum besitzt eine visitChildren()-Methode, welche die entsprechenden Visit-Methoden der untergeordneten Knoten aufruft.
Um diese rekursive Vorgehensweise verständlicher zu machen, folgt ein Beispiel.

```
/**@brief
```

```
 * verarbeitet Additionen
 */
public String visitAddition(AdditionContext ctx) {
        return visitChildren(ctx) + "\n"
                + "iadd\n";
}
```

Die Tatsache, dass es eine Methode visitAddition mit diesem Eingabeparameter und diesem Rückgabewert gibt, geht auf ANTLR zurück. Der Funktionsrumpf wurde jedoch von uns verfasst.
Zunächst werden die Kind-Knoten der Addition, sprich die Operanden, besucht. Wenn die Operanden aus weiteren mathematischen Operationen bestehen, werden zunächst diese aufgerufen, um die beliebige Länge von Ausdrücken zu ermöglichen. Handelt es sich bei einem Operanden um eine Zahl, wird die Methode visitNumber() aufgerufen.

```
/**@brief
 * verarbeitet ganze Zahlen
 */
public String visitNumber(NumberContext ctx) {
        return "ldc " + ctx.getChild(0) + "\n";
}
```

Der Befehl *ldc* steht für *load constant* und ist die Jasmin-Instruktion, um einen Wert auf den Stack zu legen. *ctx.getChild(0)* sorgt dafür, dass der Wert aus dem entsprechenden Knoten aus dem Baum entnommen wird. Der Befehl *iadd* in der visitAddition()-Methode steht für *integer addition* und ist die Jasmin-Anweisung, zwei ganzzahlige Werte vom Stack zu nehmen, diese zu addieren und schließlich das Ergebnis wieder auf den Stack zu legen.
Der Compiler-interne Ablauf für die Übersetzung des Ausdruck

```
2 + 4
```

wäre also wie folgt:

Nach der lexikalischen und syntaktischen Analyse liegt folgender Syntaxbaum vor:

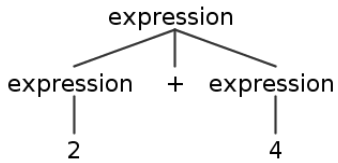


Abbildung 3: Syntaxbaum für beispielhaften arithmetischen Ausdruck

Zunächst wird die visitAddition-Methode aufgerufen, da der Operator der expression das ”+” Zeichen ist. Die visitChildren-Methode gibt nun die Zeichenkette

```
ldc 2
ldc 4
```

zurück, da die untergeordneten Expressions nur Zahlen enthalten. Dem fügt die visitAddition-Methode wiederrum den Befehl

```
iadd
```

hinzu, um die Addition durchzuführen.

In der JavaVM werden diese Instruktionen wie folgt verarbeitet: Im Initialzustand ist der Stack leer.

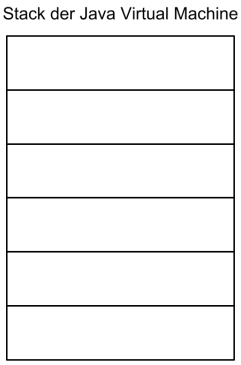


Abbildung 4: Zustand des Kellerspeicher der JavaVM

Der Befehl *ldc 2* legt den Wert *2* auf den Stack.
Der Befehl *ldc 4* legt den Wert *4* auf den Stack.
Der Befehl *iadd* nimmt zwei Werte vom Stack und legt das Ergebnis der Addition dieser Werte wieder auf den Stack

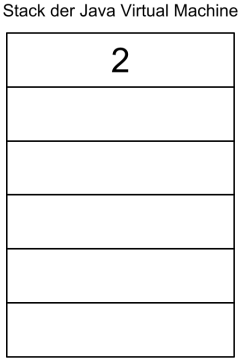


Abbildung 5: Zustand des Kellerspeicher der JavaVM

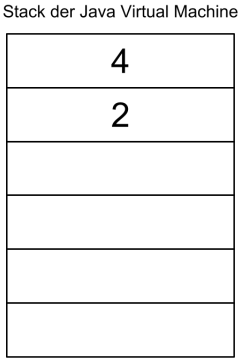


Abbildung 6: Zustand des Kellerspeicher der JavaVM

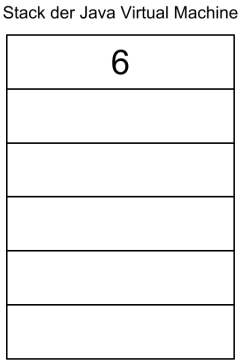


Abbildung 7: Zustand des Kellerspeicher der JavaVM

## 5.2 Ausweiten der simplen Grammatik

Nachdem die Sprache grundlegende Funktionalität erhielt, versuchten wir die weiteren Bestandteile, die durch die Anforderungen festgelegt wurden, zu implementieren.

### 5.2.1 Ergebnisausgabe

Ein Programm ohne Ausgabe ergibt keinen Sinn, da das berechnete Ergebnis nicht betrachtet werden kann. Eine Ausgabefunktion zu implementieren stellte sich hier nicht als zu großes Problem heraus, da Jasmin die Fähigkeit besitzt Objekte und Methoden der Java Library zu nutzen. Wir nutzen dies und implementierten die Ausgabefunktion wie folgt:

```
/**@brief
 * vearbeitet den Aufruf der println-Funktion
 */
public String visitPrintln(PrintlnContext ctx) {
        return  ";calling println\n" +
                //legt ein System.out Objekt auf den Stack
                "getstatic java/lang/System/out Ljava/io/PrintStream;\n" +
                //Argument der print-Funktion
                visit(ctx.argument) +
                //ruft die Methode println des System.out-Objekts auf
                "invokevirtual java/io/PrintStream/println(I)V\n\n";
}
```

Es wird zuerst ein System.out-Objekt auf den Stack gelegt, danach der auszugebende Wert. Schließlich folgt der Aufruf der Methode println(), die das Argument sowie das System.out-Objekt vom Stack wieder herunternimmt.
Der Aufruf in unserer Programmiersprache lautet:

```
println( argument );
```

## 5.3 Implementierung von Variablen

Eine Variable wird folgendermaßen deklariert:

```
int NAME;
```

Stößt der Compiler im Parsetree auf solch eine Zeile, wird geprüft, ob bereits eine Variable mit dem selben Namen im Programm existiert. Ist dies der Fall, wird eine Exception geworfen und der Kompiliervorgang bricht ab. Existiert der Name der Variablen noch nicht im Programm wird ein neuer Eintrag in der Variablentabelle erstellt.
Um einer deklarierten Variable einen Wert zuzuweisen, wird folgende Syntax verwendet:

```
NAME = EXPRESSION;
```

Wobei die EXPRESSION sich zu einem Wert evaluieren lassen muss. Wird eine solche Anweisung erkannt, wird zuerst versucht, den Namen der Variablen zu dem dazugehörigen Index der Variablentabelle auszuwerten. Gelingt dies nicht, wurde die Variable nicht deklariert und es wird eine Exception geworfen. Kann der Name der Variablen zu einem Index ausgewertet werden, wird die Variable mit

```
visit(ctx.expr)
```

auf den Stack gelegt und mit "istore [index]" der Wert der Variablen in der Jasmin-Variablentabelle gespeichert.
Eine Variable wird über ihren Namen als Teil einer Expression aufgerufen. Hier wird dann intern der Name zum Index der Variable aufgelöst. Gelingt dies nicht, wurde die Variable nicht deklariert und es wird eine Exception geworden. Ansonsten wird mit "iload [index]" der Wert der Variablen zur Verarbeitung auf den Stack der JavaVM gelegt.

## 5.4 Implementierung von Funktionen

Bevor der eigentliche Visitor arbeitet, wird der gesamte Baum auf Funktionsdeklarationen untersucht. Dies sorgt dafür, dass Funktionen an beliebigen Stellen im Quellcode definiert werden können und keine Vorwärtsdeklarationen notwendig sind. Die gefundenen Funktionen werden in einer Liste gespeichert.
Funktionen werden mit folgender Syntax deklariert:
Stößt der Compiler auf solch eine Syntax, wird anhand einer Funktionsliste geprüft, ob eine Funktion mit einer solchen Signatur bereits existiert. Ist dies so, wird eine Exception geworfen und der Kompiliervorgang bricht ab. Andernfalls wird ein neuer Eintrag mit dem Namen und der Signatur der Funktion in die Funktionsliste eingefügt.
Beim Aufruf von Funktionen wird ebenfalls geprüft, ob eine Funktion mit der gegebenen Signatur existiert. Existiert die Funktion nicht, wird eine Exception geworfen, andernfalls wird die vorher verwendete Variablentabelle gesichert und eine neue Variablentabelle für die Funktion angelegt. Dies ermöglicht die Realisierung von Variablen-Scopes. Danach werden die Argumente und Instruktionen der Funktion durch

```
invokestatic [functionname] ([ggf. parameter])
```

bearbeitet. Nach dem Abarbeiten der Funktion wird die vorherige, gesicherte Variablentabelle wieder hergestellt.

## 5.5 Implementierung von Konstanten

Konstanten sind intern als Erweiterung von Variablen anzusehen. Zusätzlich zu Variablen werden bei konstanten folgende Schritte ausgeführt: Wird eine Konstante deklariert, wird deren hasBeenAssigned-Flag auf false gesetzt. Bei der Wertzuweisung zu der Konstanten wird geprüft, welchen Wert dieses Flag besitzt. Ist der Wert false, wird eine normale Zuweisung durchgeführt und der Wert des Flags auf true gesetzt. Ist der Wert true, wurde der Konstanten bereits eine Wert zugewiesen und eine weitere Zuweisung kann nicht erfolgen. Es wird eine Exception geworfen und der Kompiliervorgang bricht ab.

## 5.6 Implementierung von Bedingten Verzweigungen

Verzweigungen sind in drei Teile aufzuteilen:

- conditionInstructions
- onTrueInstructions
- onFalseInstructions

Wird auf eine conditionInstruction getroffen, wird diese bearbeitet. Evaluiert die Expression zu true, werden die onTrueInstructions ausgeführt, die onFalseInstructions werden übersprungen. Evaluiert die conditionInstruction zu false, werden die onTrueInstructions übersprungen und die onFalseInstructions ausgeführt. Um die entsprechenden Instruktionen zu überspringen, werden Labels benötigt, die markieren, an welche Stelle des Programms gesprungen werden kann. Damit die Labels für jede Verzweigung einzigartig sind, wird gezählt, wie oft Verzweigungen vorkommen und die Labels werden entsprechend benannt.
Evaluiert werden die Bedingungen wie folgt: Ergibt die Expression 0, gilt sie als false, alle anderen Werte gelten als true. Erlaubte Operatoren der conditionInstruction-Expression sind folgende Operatoren:

- Vergleiche (<, <=, >, >=, ==)
- Logik-Gatter (& &, ||, !)

## 5.7 Implementierung einer Schleife

Eine Schleife kann intern aufgeteilt werden in

- Bedingung

- Befehle, auszuführen, wenn Bedingung wahr ist

Im Pseudocode sieht eine Schleife folgendermaßen aus:

conditionLabel
conditionInstructions
**if** *conditionInstructions == true* **then**
| Sprung zu onTrueLabel
**end**
**if** *conditionInstruction == false* **then**
| Sprung zu endOfWhileLabel
**end**
[onTrueLabel]
onTrueInstructions
Sprung zu conditionLabel
endOfWhileLabel

Die Labels werden benötigt, damit in der Schleife an die entsprechende Stelle gesprungen werden kann. Damit intern die Labels zur richtigen Schleife zugeordnet werden können, wird mitgezählt, wie oft Schleifen im Programm verwendet werden.

## 5.8 Dokumentation

Zur effizienten Erstellung der Dokumentation wurde LaTeX in Verbindung mit TexMaker verwendet. Dazu wurden die einzelnen Hauptüberschriften in eigene Dateien ausgelagert und in der Hauptdatei inkludiert. Dies hat eine erhöhte Übersichtlichkeit zur Folge. Um die Dokumentation versioniert zu speichern, wurde diese im verwendeten Git-Repository hochgeladen. Damit keine unnötigen Entwicklungs- bzw. Zwischenschrittdateien von LaTeX in das verwendete Git-Repository übernommen werden, wurde ein Alias erstellt, der ebendiese Dateien beim Alias-Aufruf automatisiert entfernt. Dieser Alias sieht wie folgt aus:

```
rmtex_dev()
{
        echo "Removing .aux"
        rm *.aux &> /dev/null
        echo "Removing .toc"
        rm *.toc &> /dev/null
        echo "Removing .synctex.gz"
        rm *.synctex.gz &> /dev/null
        echo "Removing .log"
```

```
        em *.log &> /dev/null
        echo "Removing .out"
        rm *.out &> /dev/null
}
```

# 6 Aufgetrene Probleme und deren Lösung

## 6.1 Operatorenpriorität bei arithmetischen Ausdrücken

vgl. Kapitel oben

## 6.2 Variablen

Ein wichtiger Bestandteil jeder Programmiersprache ist die Möglichkeit Variablen zu nutzen. Damit eine Variable im späteren Programmverlauf genutzt werden kann, muss sie zunächst deklariert werden. In unserer Sprache erfolgt dies durch die Angabe [Datentyp] [Name];. Beispielsweise legt der Aufruf

```
int x;
```

eine Integer-Variable mit dem Namen x an.
Als nächstes sollte eine Wertzuweisung erfolgen:

```
x = 42;
```

Nachdem diese obligatorischen Schritte vollzogen wurde, kann der Wert der Variable beliebig oft ausgelesen oder verändert werden.
Damit diese Features in die Zielsprache zu übertragen werden, nutzt unser Compiler eine Variablentabelle. Jasmin besitzt die Möglichkeit den Wert der oben auf dem Stack liegt zwischenzuspeichern. Der Befehl

```
astore <var-num>
```

nimmt den Wert vom Stack und speichert in am Index <var-num> in der Variablentabelle.
Mit dem Befehl

```
aload <var-num>
```

wird die Variable an der Position <var-nam> wieder auf den Stack gelegt. Jasmin akzeptiert nur ganze Zahlen $> 0$, jedoch keine Zeichenketten, für <var-num> Deshalb muss unser Compiler den Variablenname, den der Nutzer der Sprache wählt, zu einem Index auflösen.

### 6.2.1 Redefinition von bereits definierte Variable

Wenn im Syntaxbaum eine Variablendeklaration gefunden wird, wird zunächst überprüft, ob eine Variable mit diesem Namen bereits vorhanden ist. Für diesen Abgleich wird intern eine HashMap verwendet. Ist der Name noch

nicht vorhanden, wird er der HashMap hinzugefügt, wobei der Schlüssel der Name der Variable und der Wert die aktuelle Größe der Tabelle ist. Dies ist notwendig, um den Jasmin-Befehlen astore und aload einen Index zu übergeben.

### 6.2.2 Zugriff auf undefinierte Variable

Ein möglicher Fehler seitens des Benutzer unserer Sprache ist, dass eine Variable aufgerufen wird, obwohl diese zuvor nicht definiert wurde. Beim Aufruf einer Variablen wird deshalb zunächst überprüft, ob diese in der Variablen-Map vorhanden ist. Falls nicht wird eine entsprechende Exception ausgelöst.

## 6.3 Funktionen

### 6.3.1 Zugriff auf undefinierte Funktion

## 6.4 Redefinition von bereits definierter Funktion

### 6.4.1 Gültigkeitsbereiche

### 6.4.2 Funktionen mit gleichem Namen und unterschiedlichen Signaturen

## 6.5 Vorwärtsdeklarationen

## 6.6 Bedingte Verzweigungen

### 6.6.1 Umsetzung in Jasmin mit Hilfe von Labels und Sprungbefehlen

# 7 Verwendung der UnsereCompilerbauSprache

Ein gültiges Programm in der UnsereCompilerbauSprache besteht aus beliebig vielen Statements und Funktionsdefinitionen.

## 7.1 Datentypen

Der einzige Datentyp der Sprache ist 'int'. Ein 'int' repräsentiert eine ganze Zahl mit einer Größe von 32 bit. Mögliche Werte zur Zuweisung liegen zwischen -2147483648 und 2147483647 ($-2^{31}$ bis $2^{31} - 1$).

## 7.2 Bezeichner

Bezeichner werden genutzt, um Variablen, Konstanten sowie Funktionen zu benennen. Bezeichner müssen eindeutig sein, d.h. ein Bezeichner darf im entsprechenden Gültigkeitsbereich in nur einer Deklaration des gleichen Typ vorkommen. Gültige Bezeichner beginnen mit einem Klein- oder Großbuchstaben und dürfen von beliebig vielen weiteren Klein- bzw. Großbuchstaben sowie den Ziffern von 0 bis 9 gefolgt werden. Ein Bezeichner darf kein reserviertes Schlüsselwort sein (vgl. Liste der reservierten Schlüsselwörter).

## 7.3 Numerische Werte

Numerische Werte bestehen aus einer beliebig langen Folge der Ziffern von 0 bis 9. Ein numerischer Wert muss aus mindestens einer Ziffer bestehen.

## 7.4 Besondere Formatierungszeichen

Leerzeichen, Tabulatoren sowie Zeilenumbrüche dürfen in einer Quelltext-Datei vorkommen, werden jedoch vom Compiler ignoriert.

## 7.5 Statements

Alle Statements außer Verzweigungen, Funktionsdeklarationen und Schleifen werden mit einem Semikolon ";" beendet.

## 7.6 Variablen

### 7.6.1 Deklaration

Variablen werden mit dem Schlüsselwort *int* sowie dem gewünschten Namen nach einem Leerzeichen deklariert. Der Name wird durch einen Bezeichner

angegeben.
Beispiel:

```
int beispiel;
```

### 7.6.2 Definition

Nach dem eine Variable deklariert wurde, kann ihr ein Wert zugewiesen werden. Eine Zuweisung erfolgt durch das Gleichheitszeichen "=". Die linke Seite der Zuwesiung ist dabei der Name der Variable, die rechte Seite kann ein arithmetischer Ausdruck, der Aufruf einer Funktion, eine Variable oder eine Konstante sein. Wertzuweisungen von Variablen dürfen beliebig oft erfolgen. Die Zuweisung muss getrennt von der Deklaration erfolgen.
Beispiel:

```
beispiel = 42;
beispiel = eineFunktion();
beispiel = 1 + 2 + 3;
beispiel = eineAndereVariable;
beispiel = EINE_KONSTANTE;
```

### 7.6.3 Aufruf

Variablen dürfen auch in der rechten Seite einer Zuweisung vorkommen. Beispiel:

```
int beispiel1;
int beispiel2;

beispiel1 = 42;
beispiel2 = beispiel1 - 1;
```

## 7.7 Konstanten

Konstanten werden ähnlich wie Variablen verwendet.

### 7.7.1 Deklaration

Im Gegensatz zu einer Variablen wird dem Namen das Schlüsselwort *const int* anstatt nur *int* vorgestellt.
Beispiel:

```
const int BEISPIEL;
```

### 7.7.2 Definition

Die Wertzuweisung erfolgt wie bei einer Variable, darf jedoch nur ein mal pro Konstante erfolgen.

### 7.7.3 Ergebnisausgabe

Mit Hilfe der Funktion *println([argument])* können numerische Werte, arithmetische Ausdrücke, logische Ausdrücke, Vergleiche, Rückgabewerte von Funktionen, Variablen und Konstanten auf die Konsole ausgegeben werden.
Beispiele:

```
int beispiel;
const int BEISPIEL;

beispiel = 42;
BEISPIEL = 123;

println(42);
println(42*5);
println(42<5);
println(42<5 && beispiel==BEISPIEL);
println(testFunktion(42));
println(beispiel);
println(BEISPIEL);
```

## 7.8 Funktionen

### 7.8.1 Deklaration

Funktionen könnnen an einer beliebigen Stelle im Quellcode deklariert werden. Ein Aufruf ist auch vor der Deklaration möglich.
Eine Funktionsdeklaration setzt sich folgendermaßen zusammen: Schlüsselwort "int" für den Typ des Rückgabewerts, Bezeichner als Name der Funktion, öffnende runde Klammer, beliebig viele Variablendeklarationen (keine Übergabeparameter sind auch möglich), schließende runde Klammer und schließlich ein Funktionsrumpf. Der Funktionsrump beginnt mit einer öffnenden geschweiften Klammer gefolgt von beliebig vielen Statements. Ein besonderes Statement ist die Rückgabe. Beim Auruf von *return* wird die Funktion

verlassen. Wenn hinter *return* ein numerischer Wert, eine Variable, eine Konstante, ein arithmetischer Ausdruck, ein logischer Ausdruck oder der Aufruf einer weiteren Funktion folgt, so stellt dies den Rückgabewert der Funktion dar. Der Funktionsrumpf wird mit einer schließenden geschweiften Klammer abgschlossen.
Beispiel:

```
int add(int a, int b) {
        println(a);
        println(b);
        return a + b;
}
```

### 7.8.2 Aufruf

Funktionsaufrufe dürfen in der rechten Seite von Zuweisungen, als Argument für die println-Funktion und als Teil von logischen sowie arithmetischen Ausdrücken vorkommen. Funktionsaufrufe dürfen auch eigenständige Statements sein.
Ein Funktionsaufruf setzt sich wie folgt zusammen: [nameDerFunktion]([Parameterliste])
Die Funktion wird beim Verlassen mit dem zuvor definierten Rückgabewert substituiert.
Beispiel;

```
int x;
int y;
int z;

x = 40;
y = 2;
z = add(x, y);
```

### 7.8.3 Arithmetik

Arithmetische Ausdrücke dürfen folgende Komponenten besitzen:

### 7.8.4 Operationen

Jede Operation setzt sich aus einem linken Operanden, einem Operator sowie einem rechten Operanden zusammen.
Zulässige Operationen sind:

| Operator | Operation |
|---|---|
| + | Additionen |
| - | Subtraktionen |
| * | Multiplikationen |
| / | Divisionen |

Die Operationen sind hier in aufsteigender Bindung aufgelistet, d.h., dass beispielsweise eine Division eine höhere Bindung als eine Addition besitzt. (Umgangssprachlich "Punkt vor Strich")

### 7.8.5 Operanden

Zulässige Operanden für aritmetische Operationen sind:

- numerische Werte
- Variablen
- Konstanten
- Funktionen (bzw. deren Rückgabewerte)
- weitere Operationen

Aus dem letzten Eintrag dieser Liste ergibt sich eine Rekursion, die eine beliebige Länge von arithmetischen Ausdrücken ermöglicht.

### 7.8.6 Klammerung

Operationen dürfen Klammern mit beliebiger Verschachtelungstiefe enthalten. Geklammerte Terme besitzen die höchst mögliche Bindung, d.h. höher als eine Division.

## 7.9 Aussagenlogik

Logische Ausdrücke werden wie in C intern als "integer" gespeichert und besitzen keinen eigenen Datentyp. Dabei repräsentiert der Wert 0 den Wahrheitswert *false*, alle anderen Werte gelten als *true*. Wie auch arithmetische Ausdrücke, dürfen logische Ausdrücke eine beliebige Länge besitzen und eine beliebig tiefe Klammerschachtelung besitzen.
Logische Ausdrücke werden verwendet, um Bedingungen auszudrücken, also in if-else-statements sowie in Schleifen.
Folgende Operationen stehen dafür in dieser Priorität zur Verfügung:

| Operation | Operator | Ergebnis |
|---|---|---|
| Konjunktion | & & | Wahr, wenn beide Operanden wahr sind. Sonst false. |
| Disjunktion | \|\| | Wahr, wenn ein oder beide Operanden wahr sind. Sonst false. |
| Negation | ! | Das Gegenteil das negierten Term. |

Desweiteren stehen folgende Vergleichsoperationen zur Verfügung:

| Operator | Vergleich |
|---|---|
| < | kleiner als |
| <= | kleiner als oder gleich |
| > | größer als |
| >= | größer als oder gleich |
| == | gleich |

Wenn ein Vergleich eine wahre Aussage ist, ist das Ergebnis des Vergleich *true*. Wenn ein Vergleich eine falsche Aussage ist, ist das Ergebnis *false*.

## 7.10 Bedingte Verzweigungen

## 7.11 Schleifen

Eine Schleife beginnt mit dem Schlüsselwort *while*, wird von einer Bedingung in runden Klammern gefolgt und schließt mit einem Schleifenrumpf ab. Der Schleifenrumpf wiederrum wird mit geschweiften Klammern geöffnet und abgeschlossen.
Sowohl beim ersten Aufruf der Schleife als nach jedem Schleifendurchlauf wird überprüft, ob die gegebenene Bedingung wahr oder falsch ist. Ist die Bedingung wahr, werden die Anweisungen im Schleifenrumpf ausgeführt. Falls die Bedingung falsch ist, werden die Anweisungen nicht ausgeührt.
Beispiel (gibt die Summer der Zahlen von 1 bis 10 aus):

```
int i;
int x;

i = 0;
x = 0;

while(i <= 10) {
        i = i + 1;
        x = x + i;
}
```

```
println(x);
```

## 7.12 Reservierte Schlüsselwörter und Symbole

| Schlüsselwort | Bedeutung |
|---|---|
| int | Deklariert eine Variable |
| const int | Deklariert eine Konstante |
| println | Aufruf der Ausgabefunktion |
| return | Hinter "return" folgt der Rückgabewert einer Funktion |
| if | Beginn eines if(-else)-Statements |
| else | Beginn des else-Teil eines if-else-Statements |
| while | Beginn einer while-Schleife |

| Symbol | Bedeutung |
|---|---|
| = | Zuweisungsoperator |
| + | Operator für Additionen |
| - | Operator für Subtraktionen |
| * | Operator für Multiplikationen |
| / | Operator für Divisionen |
| < | Vergleichsoperator für kleiner als |
| <= | Vergleichsoperator für kleiner gleich |
| > | Vergleichsoperator für größer als |
| >= | Vergleichsoperator für größer gleich |
| == | Operator um zwei Werte auf Gleichheit zu überprüfen |
| & & | logisches Und |
| \|\| | logisches Oder (inklusiv) |
| ! | logisches Nicht |

# 8 Aufrufen des UnserCompilerbauCompilers

- java -jar UCC.jar sourceCode.txt ( - java -jar jasmin.jar output.j) - java output

# 9 Anhang

## 9.1 Quellenangaben

## 9.2 Quellcode des Compiler(?)