Compilerbau

Seminararbeit der ZHAW

von Marius Müller

05.Juni 2013 *Version 1.0*

Inhalt

1. Einleitung

1.1. Wie es dazu kam

Das Festlegen des Themas viel mir nicht allzu leicht. Der Grund dafür war hauptsächlich, das breit gefächerte Themengebiet.
Der Entscheid fiel schlussendlich auf das Thema Compilerbau. Dabei fielen vor allem zwei Themen ins Gewicht. Ich arbeite viel mit Java, hatte aber keine Ahnung, was mit dem von mir geschriebenen Code passiert. Des Weiteren gab es ein breites Spektrum an Möglichkeiten, dieses Thema umzusetzen.
Nach einiger Einarbeitungszeit und dem durcharbeiten des Werks „Grundlagen und Techniken des Compilerbaus" von Niklaus Wirth[1], entschloss ich mich, gegen die Beispielvorlagen des Buches, für eine eigenen Implementierung. So entstand die Idee für einen Compiler, der Javacode in die Assembler-Sprache übersetzt.
Mit dieser Idee, machte ich mich auf die Suche nach einem geeigneten Assembler, vorzugsweise einer RISC (Reduced Instruction Set Computer) Architektur. Der Grund für die Wahl dieser einfacheren Architektur ist, dass ich bisher noch keine Zeile Assemblercode geschrieben habe. Fündig wurde ich beim Prdukt „SPIM"[2]. Das Programm dazu ist kann unter http://pages.cs.wisc.edu/~larus/spim.html heruntergeladen werden.

[1] Grundlagen und Techniken des Compilerbaus, 3. Auflage von Niklaus Wirth
[2] https://www.scss.tcd.ie/John.Waldron/itral/cahome.html, 13. April 2013

2. Aufgabenstellung

2.1. Ausgangslage

Praktisch alle modernen Computerprogramme wurden in Hochsprachen wie Java, .Net oder C# entwickelt. Doch was passiert mit diesem Code? Im Normalfall beschränkt sich das Wissen gerade noch darauf, dass man weiss, dass ein Compiler das Programm übersetzt, damit der Computer die geschriebenen Zeilen ausführen kann. In der Arbeit werde ich diesen Vorgang unter die Lupe nehmen.

2.2. Ziele

Das Ziel der Arbeit ist es, einen Blick in die Welt des Compilerbaus zu erhalten. Das Buch „Grundlagen und Techniken des Compilerbaus" von Niklaus Wirth dient dazu einen ersten Blick auf dieses Thema zu werfen. In einem zweiten Schritt wird ein in Java geschriebener Compiler erarbeitet. Dieser soll in der Lage sein, eingelesenen Java-Code in ein Format zu übersetzen, welcher von PCSpim erkannt wird und ausgeführt werden kann. Der Benutzer des Programms kann ausser der Eingabe, keinen Einfluss auf den Compiler nehmen. Des Weiteren ist auch keine grafische Oberfläche dafür geplant.

2.3. Erwartete Resultate

Die Dokumentation der Arbeit erläutert die Erkenntnisse, die während der Einarbeitungsphase gemacht wurden. Ausserdem wird auf das geschriebene Programm eingegangen. Dieses ist der Kern der Arbeit und ebenfalls Teil des Resultates.

2.4. Termine

Datum	Ziel
20. März	Kickoff-Meeting
5. Juni	Abgabe der Arbeit
12. Juni	Präsentation der Arbeit

3. Projektplanung

Projektphase	Soll (in h)	Ist (in h)
Analyse		
Projekt definieren	2	2
Projektplan definieren und erstellen	2	2
Entwicklung		
Einarbeiten in das Thema Compilerbau anhand des Buches „Grundlagen und Techniken des Compilerbaus“	10	12
Einarbeiten in die Assemblerprogrammierung	5	7
Scanner-Funktion einbauen	5	3
Parser implementieren	15	20
Codegenerator implementieren	10	10
Testing		
Fortlaufendes Testing & funktionales Testing	3	4
Abschliessendes Testing	3	1
Dokumentation		
Projektdokumentation	10	10
Präsentation	4	3
Total	69	74

4. Anforderungen

Da alle Anforderungen von Anforderung F11 abhängig sind, wurde dieser Punkt jeweils unter „Abhängigkeit zu" weggelassen. Der Grund für das späte Beenden dieses Punktes liegt darin, dass zu Beginn der Implementierungsphase nur das Lesen aus einem File abgearbeitet wurde.

Id: F1	**Version:** 1.0.0	**Prio:** 1
Titel Werte zuweisen **Beschreibung** Der Compiler muss in der Lage sein, Integer, Boolean und String Parameter zu erkennen und den Code zur Erstellung, Initialisierung und Wertezuordnung zu generieren.		
Abhängigkeit zu:	**Datum:** 12.05.2013	**Status:** Beendet

Id: F2	**Version:** 1.0.0	**Prio:** 1
Titel: Mathematische Ausdrücke **Beschreibung:** Der Compiler muss in der Lage sein, einfache, ganzzahlige Rechenaufgaben mit zwei Parameter oder Konstanten zu erkennen und den Code für dessen Berechnung auszugeben		
Abhängigkeit zu: F1	**Datum:** 18.05.2013	**Status:** Beendet

Id: F3	**Version:** 1.0.0	**Prio:** 1
Titel: Konsolenausgabe **Beschreibung:** Der Compiler muss in der Lage sein, den Code zu generieren, welcher Werte aus Integer und String-Variablen liest und diese in der Konsole ausgibt.		
Abhängigkeit zu: F1	**Datum:** 23.05.2013	**Status:** Beendet

Id: F4	**Version:** 1.0.0	**Prio:** 1
Titel: Klassenkontext erkennen **Beschreibung:** Der Compiler muss in der Lage sein, die Klassen- und Methodenhülle zu erkennen und diese Korrekt zu generieren		
Abhängigkeit zu:	**Datum:** 25.05.2013	**Status:** Beendet

Id: F5	**Version:** 1.0.0	**Prio:** 1
Titel: Methodenaufrufe **Beschreibung:** Der Compiler muss in der Lage sein, Methodenaufrufe zu erkennen und den Code für den Sprung zur Methode, sowie den Rücksprung zur ursprünglichen Methode zu generieren.		
Abhängigkeit zu: F4	**Datum:** 25.05.2013	**Status:** Beendet

Id: F6	**Version:** 1.0.0	**Prio:** 1
Titel: Vergleiche **Beschreibung:** Der Compiler muss in der Lage sein, den Code, mit dem zwei Integer miteinander verglichen und den Wahrheitsgehalt des Vergleichs abgespeichert wird, zu erkennen und zu interpretieren.		
Abhängigkeit zu: F1	**Datum:** 26.05.2013	**Status:** Beendet

Id: F7	**Version:** 1.0.0	**Prio:** 1
Titel: If-/else-Bedingungen **Beschreibung:** Der Compiler muss in der Lage sein, if/else if/else Bedingungen korrekt zu erkennen und den Code dafür zu generieren.		
Abhängigkeit zu:	**Datum:** 30.05.2013	**Status:** Beendet

Id: F8	**Version:** 1.0.0	**Prio:** 2
Titel: Loops **Beschreibung:** Der Compiler muss in der Lage sein, while-Schleifen zu erkennen und den Code dafür zu generieren.		
Abhängigkeit zu: F5, F7	**Datum:** 30.05.2013	**Status:** Beendet

Id: F9	**Version:** 1.0.0	**Prio:** 2
Titel: Lesen aus der Konsole **Beschreibung:** Der Compiler muss in der Lage sein, den Befehl für das Lesen aus der Konsole zu erkennen und den Code dafür zu generieren		
Abhängigkeit zu: F1	**Datum:** 01.06.2013	**Status:** Beendet

Id: F10	**Version:** 1.0.0	**Prio:** 3
Titel: Kommentare ignorieren **Beschreibung:** Der Compiler muss in der Lage sein, Kommentare im Java-Code zu ignorieren und mit dem ersten Befehl nach dem Kommentar weiterzufahren.		
Abhängigkeit zu:	**Datum:** 03.06.2013	**Status:** Beendet

Id: F11	**Version:** 1.0.0	**Prio:** 1
Titel: Lesen und schreiben aus/in Files **Beschreibung:** Das Programm muss in der Lage sein, ein Source-Code-File einzulesen, den Text darin dem Compiler zur Verfügung zu stellen und zum Schluss den generierten Code in ein weiteres File zu schreiben		
Abhängigkeit zu:	**Datum:** 03.06.2013	**Status:** Beendet

5. Compiler

5.1. Geschichtliches

Der erste Compiler wurde 1956 für die Sprache Fortan (Formula translator) gebaut.
Im Jahre 1960 folgte dann mit Algol 60, die erste Sprache mit einer systematischen Struktur. Anhand von dieser konnten die Grundlagen der Technik des Compilerbaus erarbeitet werden, welche noch heute gelten.
Bis ca. 1980 wurden sogenannte Mehrphasen-Compiler eingesetzt. Dabei wurde eine Phase des Übersetzungsprozesses abgearbeitet und das Resultat zwischengespeichert. Dieses Resultat diente dann als Grundlage für die nächste Phase. In der Regel wurden zwischen vier und sechs Phasen eingesetzt. Allerdings gab es auch Fälle bei der bis zu 70 Phasen notwendig waren.
Weil der Arbeitsspeicher immer leistungsfähiger wurde, konnte schliesslich diese Methode überarbeitet werden, da dadurch die Zwischenspeicherung weggelassen werden konnte. Aus diesem Grund wurde ein Einphasen-Compiler entwickelt. Aufgrund des wegfallens des oben erwähnten Teilschrittes, konnte dieser eine deutlich bessere Performance aufweisen.
Bei heutigen Compilern wurde diese eine Phase aber wieder auf zwei Teile aufgeteilt, den Frontend und den Backend. Während die Frontend-Phase für die Sprache gilt und deshalb nur einmal implementiert werden muss, existiert für jedes Betriebssystem ein speziell dafür angefertigtes Backend.
Ein Spezialfall sind die Embedded Systems. Da bei ihnen der Speicher meist nicht ausreicht um einen Compiler aufzunehmen, werden die auszuführenden Codes im Normalfall von einem fremden Rechner erzeugt. Diesen Compiler wird Cross Compiler genannt.

5.2. Analyse von Kontextfreien Sprache mittels Bottom-Up-Parsing (LR-Parsing)

Das Ziel dieser Parsing-Methode liegt darin, zu zeigen, dass der zu bearbeitende Text dem Startsymbol entspringt. Alle in der Syntax angetroffenen Nichtterminalsymbole werden als Unterziele aufgefasst. Das Resultat dieses Parsings ist ein Syntaxbaum, bei dem das Startsymbol die Wurzel ist.
Um dies zu erreichen besitzt der Parser zwei Möglichkeiten. Erstens, er liest ein weiteres Symbol des Textes, der Lesekopf wird also verschoben. Als Alternative kann er eine bereits gelesene Symbolfolge, in ein einziges Nichtterminal-Symbol, durch Anwendung der Syntaxregel, ersetzen.

Beispiel zum Vorgehen des LR-Parsings, dargestellt anhand des Satzes (a+b)*c, wobei die Art des Schrittes mit S(=shift) und R(=reduce) angegeben wird:

Syntax:
E = T | E “+” T
T = F | T “*” F
F = id | “(“ E “)”

Schritt	Parsingstatus	Restliche Eingabe
S	(	a+b)*c
S	(a	+b)*c
R	(F	+b)*c
R	(T	+b)*c
R	(E	+b)*c
S	(E+	b)*c
S	(E+b	)*c
R	(E+F	)*c
R	(E+T	)*c
R	(E	)*c

S	(E)	*c
R	F	*c
R	T	*c
S	T*	C
S	T*c	
R	T*F	
R	T	
R	E	

Das Problem beim LR-Parsing liegt darin, zu wissen, von welcher Art der nächste Schritt sein soll. Des Weiteren ist jeweils zu entscheiden, wie weit eine anstehende Reduktion im Stack nach links reichen soll, also welche Symbole miteingeschlossen werden sollen.
Um ein effizientes Parsing anbieten zu können, arbeitet der Bottom-up-Parser immer mit Tabellen.

6. Aufbau eines Compilers

Der Compiler ist in drei Phasen unterteilt. Der Scanner, welcher die Codesequenzen ihrer jeweiligen Eigenschaft zuweist, und der Parser, der mit den, vom Scanner erhaltenen, Informationen weiterverarbeitet, gehören zum Frontend des Compilers, während der Codegenerator zum Backend gehört.

6.1. Scanner

Der Scanner erkennt in einem Quelltext die Terminalsymbole. Zu dieser Menge gehören Bezeichner, Zahlen und ein Vokabular. Dieses muss als erstes, noch bevor die erste Implementierung erfolgt festgelegt werden.
Erkannte Symbolwerte werden auf ganze Zahlen abgebildet, die durch die Konstantendefinition festgelegt werden.
Bei diesem Verfahren werden zwei Prozeduren aufgerufen. Die Prozedur *Mark* dient zum Erkennen von Fehlern. Wurde ein Fehler gefunden wird dessen Textposition im Quelltext markiert.
Die wichtigere Prozedur des Scanners ist aber das *Get*. Diese erfüllt folgende Aufgaben:

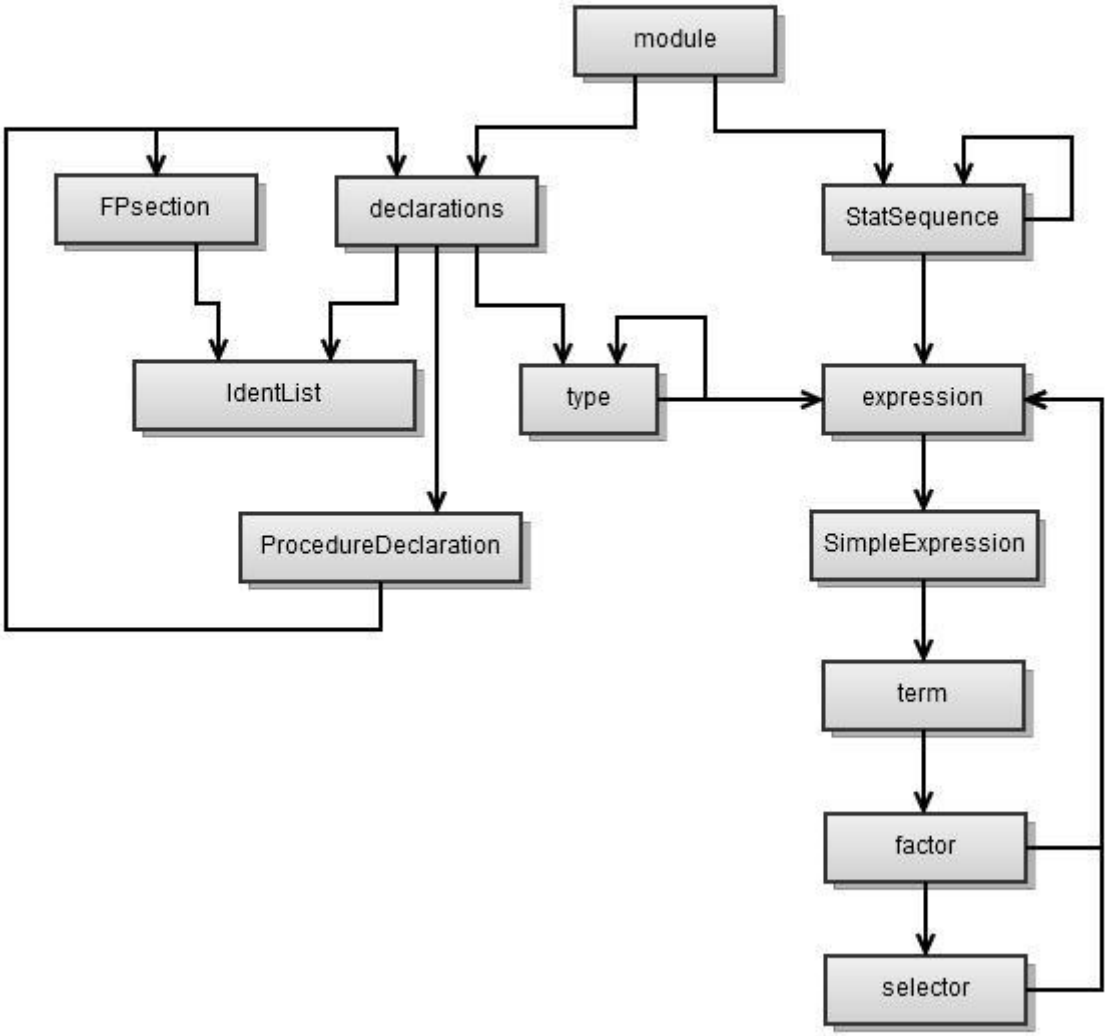
1. Leerzeichen und Zeilenenden werden übersprungen.
2. Reservierte Wörter werden erkannt.
3. Nicht reservierte Wörter werden als Bezeichner erkannt. Diese werden der globalen Variable *id* zugewiesen. Der Parameter *sym* erhält den Wert „ident".
4. Zahlen werden erkannt und diese der globalen Variable *val* zugewiesen. Der Parameter *sym* erhält den Wert „number".
5. Kombinationen von Spezialzeichen (z. B. +=) werden als ein Symbol erkannt.
6. Kommentare werden übersprungen
7. Symbol *null* wird erzeugt, wenn der Scanner ein illegales Zeichen erkennt. Das Symbol *eof* wird erzeugt, wenn das Ende eines Textes erreicht wird.

6.2. Parser

Als erstes muss die Mengen der Anfangs- und der Folgesymbole ermittelt und definiert werden. Folgende Tabelle soll nur zur Anschauung dienen und ist nicht komplett:

S	**First(S)**
Factor	(int indent
Term	(int indent
SimpleExpression	+- (int indent
ProcedureCall	Indent
Statement	Ident IF IF ELSE WHILE
FieldList	Ident
FPSection	ident VAR
MethodHeading	"private static void" public static void"
MethodBody	{]}
S	Follow
Selector	* / + - == != < <= >= >) { } = "else if"
Factor	* / + - == != < <= >= >) { } "else if"
Term	+ - == != < <= > >=) { }

Die Syntax mit der Methode des rekursiven Abstiegs kann mit einem Lookahead von einem einzigen Symbol bearbeitet werden. Dabei entspricht jedes Nichtterminalsymbol einer Prozedur. Bevor die Prozeduren des Parsers formuliert werden, sollte festgestellt werden, wie die einzelnen Prozeduren aufeinander Bezug nehmen. Hierfür wird ein Bezugsdiagramm gezeichnet.



Jede Prozedur ist als Knoten dargestellt. Von diesen aus sind Verweise auf diejenigen Prozeduren, auf welche sich der Knoten direkt oder indirekt abstützt.
Beim Bezugsdiagramm werden bewusst gewisse Nichtterminalsymbole weggelassen, da diese in anderen Symbolen eingeschlossen sind (z.B. ArrayType in type).
Jede Schleife in diesem Diagramm beschreibt eine Rekursion. Es muss also sichergestellt werden, dass für die Programmierung eine Sprache zur Verfügung gestellt wird, die rekursive Aufrufe von Prozeduren zulässt.
Das einzige Modul, auf das nie Bezug genommen wird, ist das Startsymbol. Der Befehl compile setzt den Parsing-Prozess durch Aufruf dieser Prozedur in Gang.

6.2.1. Behandlung von syntaktischen Fehlern

Sobald dem Parser ein inkorrektes Symbol vorliegt, wird der Vorgang der Satzzerlegung abgebrochen und der Parser hat seine Aufgabe erfüllt.
In der Praxis ist dieses Vorgehen natürlich unbrauchbar, da man dabei keinerlei Angaben über den Ort des Abbruchs erhält. Vielmehr muss der Compiler in der Lage sein, den Fehler mit weiteren Angaben (wie Ort usw.) auszugeben und mit der Zerlegung fortzufahren.
Um dies zu erreichen, wird anhand folgender Kriterien eine Strategie für die Fehlerbehandlung zurechtgelegt:

1. Es sollten so viele Fehler wie möglich in einem Durchlauf erkannt werden.

2. Sie soll so mit wenigen Zusatzangaben über die Syntax wie möglich auskommen.
3. Die Analyse von fehlerfreiem Text darf nicht gebremst werden.
4. Parser soll nicht unnötig aufgebläht werden.

Grundsätzlich wird zwischen zwei Fehlertypen unterschieden. Fehlende Symbole: Wenn zum Beispiel am Ende eines Faktors eine abschliessende Klammer fehlt, reicht es, wenn eine einfache Fehlermeldung angezeigt wird und danach mit der Satzanalyse fortgefahren wird.
Beim zweiten Fall, treten falsche Symbole auf. Diese müssen entweder ersetzt oder übersprungen werden. Die Analyse wird nach einem solchen Vorfall, an einem späteren Zeitpunkt im Text fortgesetzt. Dabei wird eine Ordnung unter Symbolen vorausgesetzt. Starke Symbole, die nicht übersprungen werden dürfen haben dabei eine hohe Ordinalzahl.
Im Normalfall leitet der Parser nach der Methode des rekursiven Abstiegs die Syntax her. Bei einem fehlerhaften Symbol wird in der Regel ein Fehler angezeigt und die Analyse wird bis zur nächsten Synchronisationsstelle fortgesetzt. In der Regel werden dadurch weitere Fehler angezeigt, die sich aber meist als Folgefehler herausstellen.
Das wesentliche Merkmal eines guten Compilers ist, dass zum einen keine Symbolfolgen zum Zusammenbruch des Compilers führen kann und zum anderen, dass häufige Fehler korrekt diagnostiziert werden und möglichst wenige Folgefehler verursacht werden.

6.2.2. Einbezug von Kontext durch Deklaration

Programmiersprachen sind zwar auf einer kontextfreien Syntax aufgebaut, im allgemeineren Sinn, sind sie aber trotzdem kontextabhängig. So wird jeder Bezeichner in der Deklaration, einem Objekt zugeordnet. Diesem Objekt werden wiederum Attribute, wie zum Bespiel der Datentyp, zugewiesen. Die Eigenschaften des Bezeichners liegen also im Kontext des Konstrukts.
Der Kontext wird durch eine Symbol-Tabelle verkörpert. Jeder Eintrag assoziiert den Bezeichner mit seinem Objekt. Die Bezeichnung Symbol-Tabelle stammt von Assembler ab, bei der die Bezeichner als Symbole bezeichnet wurden.
Der Parser muss so erweitert werden, dass beim Erkennen einer Vereinbarung, die Tabelle mit dem definierten Bezeichner erweitert wird.
Ein typisches Attribut dieser Tabelle ist die Objektklasse, deren Aufgabe es ist, den Bezeichner als Konstante, Variable, Typen oder Prozedur erkennen zu geben.

6.2.3. Einträge von Datentypen

Sofern die Sprache Datentypen kennt, ist eine der Kernaufgaben eines Compilers, die Konsistenzprüfung solcher. Diese Prüfung basiert auf dem Attribut type der Objekte. Da Datentypen ebenfalls deklarierbar sind, reicht ein Verweis auf das betreffende Objekt der Klasse Type. Wenn der Typ der Variable aber Anonym bleibt, funktioniert dies nicht mehr. Deshalb sollte ein neuer Datentyp eingeführt werden, der im Compiler die Darstellung der Typen übernimmt.

6.3. Codegenerator

Der Compiler besitzt für jedes Betriebssystem einen anderen Codegenerator. Bei der Codeerzeugung, existieren zwei Varianten, die direkte und die verzögerte Codeerzeugung.
Die direkte Codeerzeugung bedient sich einem Stack. Dies ist auch der Grund, warum sich diese Variante nicht durchgesetzt hat. Das Problem ist dabe, dass auf die einzelnen Register nicht mehr zugegriffen werden konnte. Diese einfache Codegenerierung erzeugt zwar korrekten, aber nicht optimalen Code. So müssen zum Beispiel Konstanten nicht in ein Register geladen werden. Der Rechner besitzt Befehle, die solche Konstanten direkt zu Registerwerten addieren können. Dies wird mit der verzögerten Codegenerierung bewerkstelligt.

Dessen Vorgehen besteht darin, die Information direkt dem Konstrukt hinzuzugeben und nicht zuerst als Code auszugeben. Dies kann mittels attribuierten Syntax erreicht werden. Dabei wird berücksichtigt, dass die Codeausgabe nicht nur von den syntaktisch zu reduzierenden Symbolen abhängt, sondern auch von deren Attributen. Es wird also ein Parameter hinzugefügt, der in Parser-Prozeduren die Attribute verkörpert. Für diesen Parameter wird die Record-Struktur gewählt, da es sich meistens um zusammengesetzte Werte handelt. Der Parametertyp wird Item genannt.
Um entscheiden zu können, ob der Wert eines Faktors, Terms oder Ausdrucks in ein Register geschrieben werden soll oder ob es sich dabei um eine Konstante handelt, muss ein Attribut mit dem Modus angegeben werden.

Item-Modus	Objektklasse	Adressmodus	Zusätzliche Attibute
Var	Var	Direct	a Wert liegt im Speicher an Adresse 'a'
Const	Const	Immediate	a Wert ist Konstante 'a'
Reg	-	Register	r Wert liegt im Register R[r]

Somit ist der Datentyp *Item* also ein Record aus den Feldern *mode, type, a* und *r*. Der Datentyp des Konstrukts ist stets ein Integer. Die Parser-Prozeduren sind somit eigentlich Funktionen vom Typ *Item*.
Der grosse Vorteil der verzögerten Codeausgabe liegt darin, dass diese erst ausgeführt wird, wenn klar ist, dass es keine bessere Lösung gibt. Das heisst, erst zu diesem Zeitpunkt wird ein Operand in ein Register geladen.

6.3.1. Datenpräsentation zur Laufzeit

Die Information, wie Daten im Speicher dargestellt werden, kann bei jeder Zielarchitektur zu unterschiedlich sein. Allerdings wird dies mittlerweile auf fast allen Zielmaschinen ähnlich gehandhabt. Der Speicher stellt immer eine Sequenz von Byte-Zellen dar, wobei alle dieser Zellen adressiert sind. Variablen, welche aufeinanderfolgend deklariert wurden, werden auch sequentiell alloziert.
Jeder Rechner bietet gewisse Datentypen an, was auch bedeutet, dass er Operationen auf diesen Typen anbietet. Die angebotenen Typen sind skalare, elementare Typen, die eine kleine Sequenz aufeinanderfolgender Bytes belegen.
Jede Variable besitzt eine Adresse und jeder Datentyp besitzt eine Grösse. Beim Betrachten eines Arrays liegen folgende Informationen daher auf der Hand: die Grösse des Arrays ist n*size(T0), als Anzahl Attribute mal die Grösse des Typen T0 und die Position eines Attributes an der Position x lässt sich mit der Adresse des Arrays plus x mal der Grösse des Typen T0 herleiten.

Die absoluten Adressen der Objekte sind zum Zeitpunkt der Kompilierung noch unbekannt. Alle zugeordneten Adressen sind relativ zu einer Basisadresse, welche erst zur Laufzeit definiert wird. Die effektive Adresse ist die Summe aus der Basis und der vom Compiler berechneten Feldadresse.
Bei einer Byte-Adressierung sollte darauf geachtet werden, dass Variablen nicht auf mehere Worte aufgeteilt wird. Diese Splittung hätte zur Folge, dass ein Variablenzugriff mehrere Speicherzugriffe nach sich ziehen würde und die wiederum hätte Einflüsse auf die Effizienz. Um dies einfach zu verhindern, sollte bei der Vergabe jede Adresse auf das nächste vielfache der Grösse des Variablentyps aufgerundet werden.

Var a: Char; b, c: Integer; d: Real

Nicht aliniert aliniert

6.3.2. Bedingte und wiederholte Anweisungen, Boole'sche Ausdrücke

6.3.2.1. Vergleiche und Sprünge

Bedingte oder auch wiederholte Anweisungen werden mittels Sprungbefehlen implementiert. Meist sind die Rechner nicht compilerfreundlich. Aus diesem Grund kann meist ein Vergleich nicht durch einen boole'schen Wert ersetzt werden. In der Regel wird ein Vergleich durch eine Subtraktion ersetzt und das Resultat liefert zurück, ob die Differenz positiv, null oder negativ ist. Dieses Ergebnis wird meist in einem speziell dafür angelegten Register, dem Condition Code, dargestellt. Das Register besteht nur aus vier Bits, N, Z, C und V genannt.

6.3.2.2. Bedingte und wiederholte Anweisungen

Die Schwierigkeit bei Compilern mit nur einem Durchlauf des Quelltextes ist, dass beim Zeitpunkt der Sprunginstruktion, die Zieladresse noch nicht bekannt ist. Dies kann gelöst werden, indem bei der Ausgabe des Sprungs, deren Adresse dem beschreibenden Item beigegeben wird. Diese wird zum Zeitpunkt, an dem die Zieladresse bekannt ist, wieder benötigt. Diese Methode ist nur möglich, da der Code als Array gespeichert wird und dadurch einzelne Instruktionen korrigiert werden können. Sprungbefehle geben immer die Anzahl Befehle an, die übersprungen werden.
Die Handhabung einer While-Schleife ist ähnlich wie bei einer If-Anweisung. Neben dem bedingten Vorwärtssprung, ist ein unbedingter Rückwärtssprung zum Anfang der Schleife notwendig.

6.3.2.3. Boole'sche Operationen

Obwohl Boole'sche Ausdrücke gleich wie arithmetische Ausdrücke aufgebaut sind und deshalb auch genauso behandelt werden können, wäre dies ineffizient. So muss zum Beispiel bei einer OR-Operation der zweiter Operand gar nicht mehr ausgewertet werden, wenn der erste erfüllt ist.
Bei Programmiersprachen, wäre die erzwungene Auswertung bieder Operanden, bei Erfüllung des Ersten sogar komplett falsch. Der Grund dafür ist, dass der zweite Ausdruck auch undefiniert sein darf und dieser Zustand vom ersten Operand "geschützt" wird.
Das wohl bekannteste Beispiel hierfür in Java ist die Überprüfung, dass eine String-Variable nicht leer sein darf. Der Test dafür sieht wie folgt aus: "str == null || str.equals("")". Im ersten Schritt wird überprüft, ob das Objekt str überhaupt definiert ist. Im zweiten ob es sich um ein leeres String-Objekt handelt. Würden immer beide Operationen überprüft, würde, falls die Erste erfüllt, die Zweite ein Fehler zurückgeben, da die Methode equals für ein null-Objekt nicht bekannt ist.

7. Projekt

7.1.1. Projektentscheid

Für die praktische Arbeit kristallisierten sich zwei Möglichkeiten heraus. Entweder wird der Compiler in Oberon-0 erfasst, da die ganzen Beispiele im Buch auf dieser Sprache beruhen oder es findet eine Loslösung vom Buch statt und der Compiler wird in der bereits bestens bekannten Java-Sprache erfasst.
Anhand folgender Punkte wurde der Entscheid für eine eigene Java-basierte Lösung gefällt.

Punkt	Punkte	Compiler in Oberon-0	Compiler in Java
Leitung durch Quellen	3	+	-
Erfahrungen	3	-	+
Freiheit	2	0	+
Vorhandene Entwicklungsumgebungen	3	0 (keine bekannt)	+
Hilfe im Web	1	0	+
Total		**0**	**6**

7.2. Projektumfang

Die Arbeit musste auf eine gewisse Anzahl von implementierten Java Befehlen reduziert werden. Dabei wurde geachtet, dass möglichst verschiedene Möglichkeiten implementiert wurden. So wurde auf Gleitkomazahlen als Ganzes verzichtet. Dies aufgrund der Tatsache, dass diese von der Logik her sich fast gleich wie ganzzahlige Werte verhalten. Zudem wurden weitere Befehle aufgrund ihrer Komplexität weggelassen. Hierbei sind unter anderem String Vergleiche und Manipulationen gemeint.
Um den Compiler ausführen zu können, muss an erster Position ein Inputfile und an zweiter ein Outputfile angegeben werden, wobei das Zweite nicht existieren muss.

7.3. Entwicklung

7.3.1. Scanner

Aufgrund der Tatsache dass ein Compiler realisiert wurde, der Java-Code in Assemblercode umwandelt, war das Vokabular bereits vorgegeben und musste nicht nochmals festgelegt werden. Da aber nicht alle Befehle des riesigen Java-Befehlssatzes berücksichtigt werden konnten, sind hier die implementierten aufgelistet:

Befehl	Beschreibung
public class	Klassendefinition
public static void	Methodendefinition
System.out.print	Konsolenausgabe
System.out.println	Konsolenausgabe mit new Line
BufferedReader	Kennzeichnet die Konsolen-Read-Variable
public	Wird ignoriert
private	Wird ignoriert
String	String-Definition
int	Integer-Definition
boolean	Boolean-Definition
if	Überprüfung eines Wahrheitswertes
else if	Alternative Überprüfung eines Wahrheitswertes
else	Alternative, wenn keiner der direkt vorangehenden

	Wahrheitswerte korrekt war
true	Wahr (wird zu einer 1 umgewandelt)
false	Falsch (wird zu einer 0 umgewandelt)
<; <=; ==; !=; >=; >	Vergleichsoperatoren
+; -; *; /	Mathematische Operatoren
;	Befehl abschliessen
{ }	Start/Ende eines abgeschlossenen Bereiches
( )	Abgrenzung einer Anweisung
/*; */	Beginn und Ende eines Kommentars

Da sehr viele gute Tools zum Schreiben von Java-Code existieren (wie Eclipse, NetBeans usw.), die Fehler bereits währen der Entwicklungszeit erkennen, wurde auf die Prozedur *Mark* verzichtet und das Augenmerk ganz auf die Prozedur *Get* gerichtet.

7.3.2. Parser

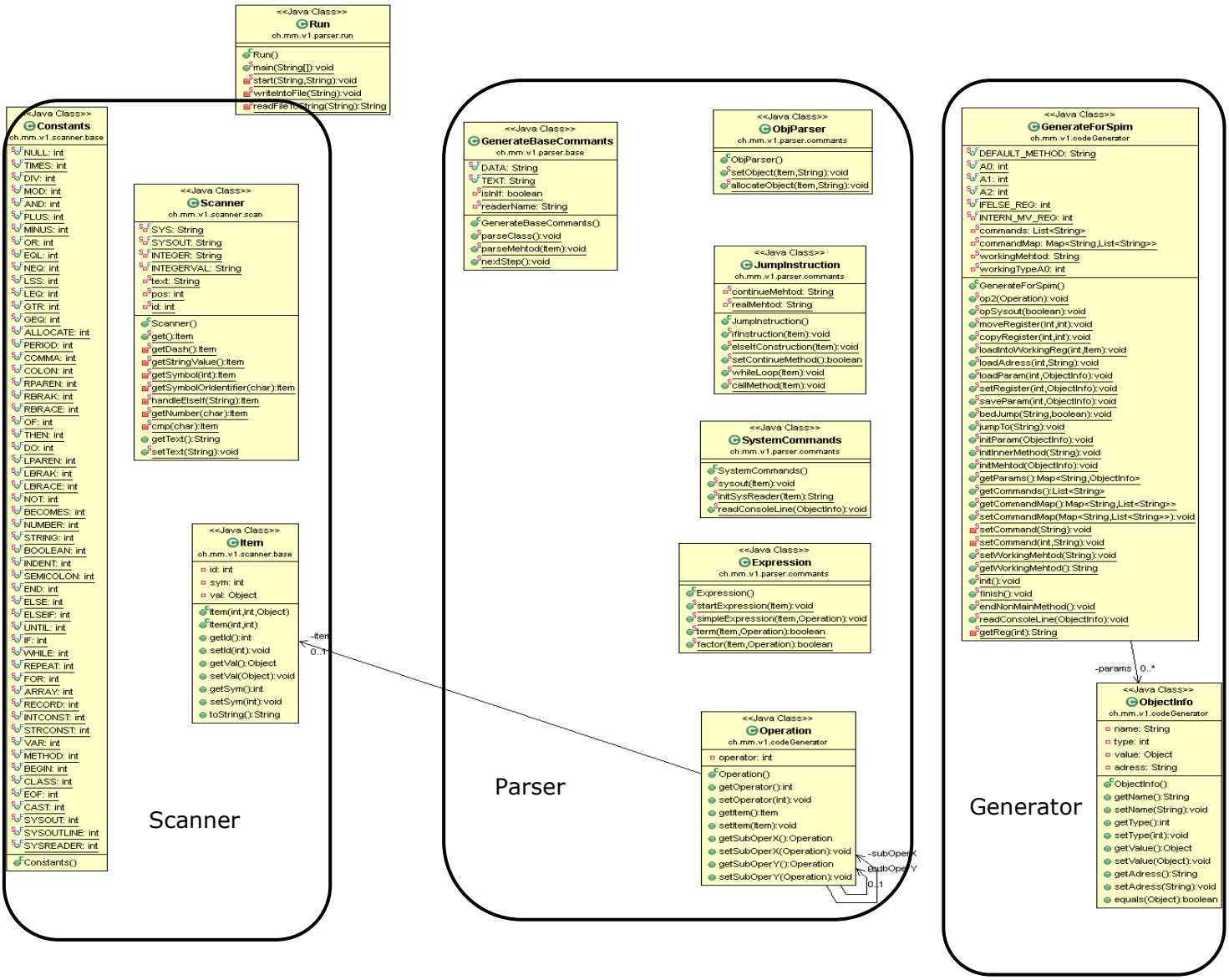
Das Herzstück des Compilers ist der Parser. Analog zum Scanner ist auch die Menge der Anfangs- und Folgesymbole gegeben. Zudem wurde, wie im Scanner erwähnt, auf die Fehlererkennung verzichtet.
Auch bei der Parser-Logik mussten einige Einschränkungen gemacht werden. Die Möglichkeiten und Einschränkungen sehen wie folgt aus:

- Es kann immer nur eine Klasse übersetzt werden und diese darf keine Abhängigkeiten zu anderen Klassen aufweisen
- Alle Parameter werden als globale Parameter gespeichert
- Methodenaufrufe sind möglich, erlauben aber keine Input und Return Parameter
- Kommentare sind als „/* */" Gruppen erlaubt, nicht erlaubt sind „--" Kommentare
- Strings können von der Konsole eingelesen und ausgegeben werden. Zudem ist es möglich String-Zuweisungen vorzunehmen
- Ein String muss als Object definiert werden, um in der Konsolen ausgegeben werden zu können (`System.out.println("bsp");` ist also nicht möglich)
- Integer Werte dürfen als Parameter, Konstante oder auch Expression ausgegeben werden
- Integer Werte können von der Konsole eingelesen (im Format `int i = Integer.valueOf(<<name>>.readLine());`) und ausgegeben werden. Zudem ist es möglich Integer-Zuweisungen vorzunehmen.
- Der Console-Reader muss wie folgt definiert werden:
  - `BufferedReader <<name>> = new BufferedReader(new InputStreamReader(System.in));`
- Mit Integer Werte können folgende Operationen vorgenommen werden
  - Mathematische Operationen: Addition, Subtraktion, Multiplikation und Division
  - Vergleiche: kleiner als, kleiner gleich, gleich, ungleich, grösser gleich und grösser als
- Logische Operationen: if, if else und else
- Schleifen: while-Schleifen; Andere Schleifen wurden nicht implementiert, da diese durch eine Äquivalente while-Schleife abgebildet werden können.
- Try Catch-Blocks sind nicht erlaubt. Das Exceptionhandling muss mit throw (die ignoriert werden) bewerkstelligt werden.
- Es kann nicht mit Listen und Arrays gearbeitet werden

7.3.3. Codegenerator

Der Codegenerator nimmt die vom Parser erhaltenen Befehle entgegen und schreibt die dazugehörigen Assembler-Befehle. Diese sind im Kapitel Assembler genauer beschrieben.

7.4. Klassendiagramm

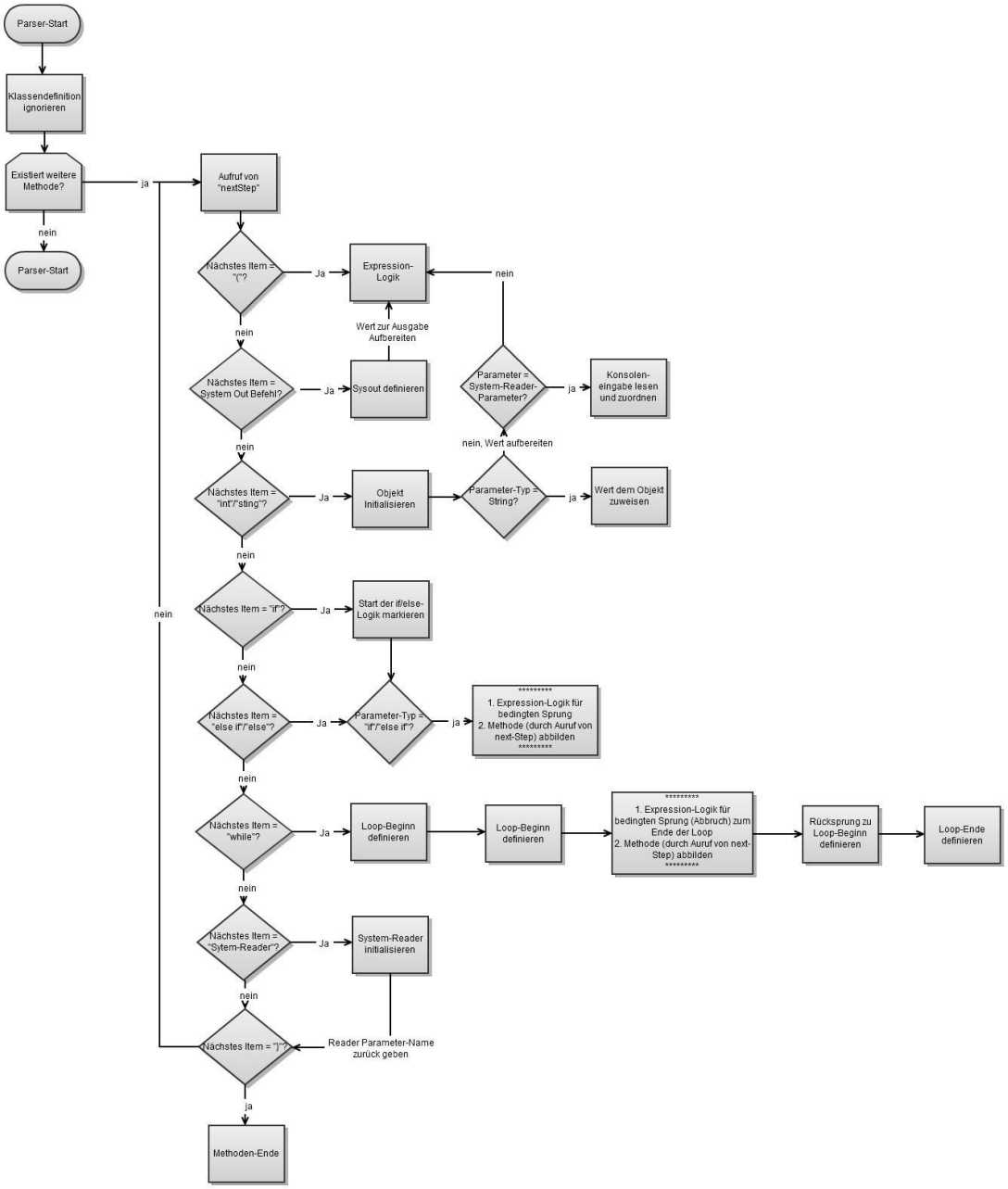


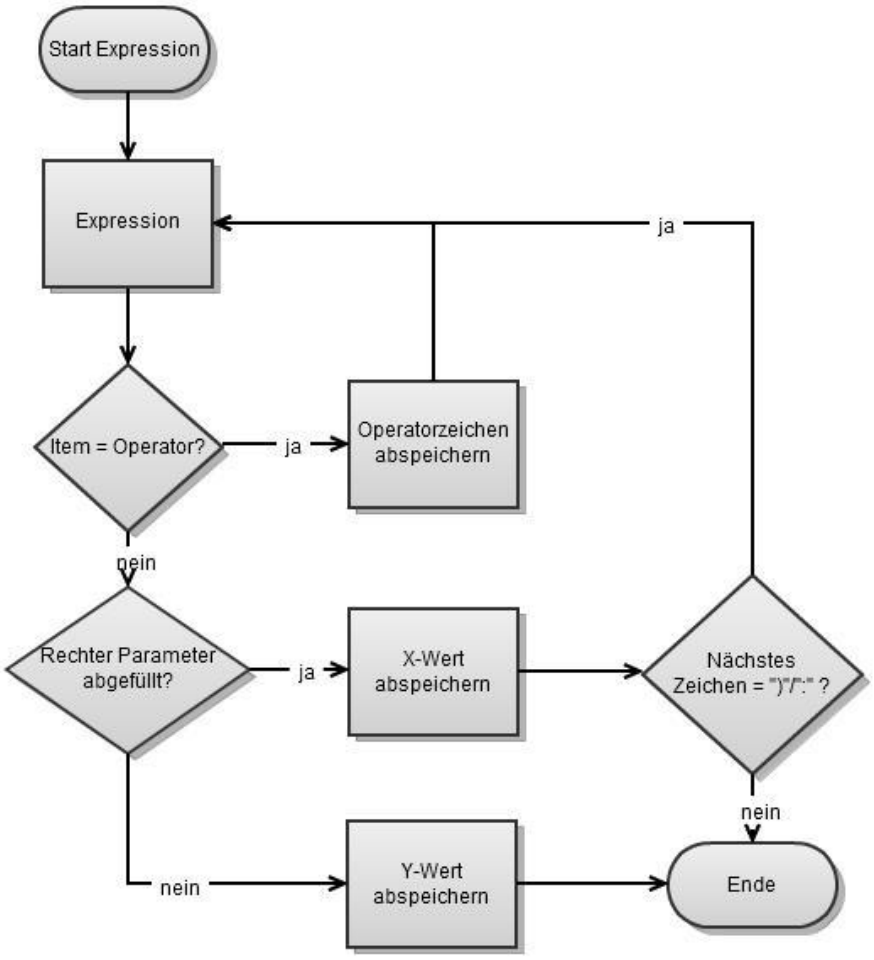
7.5. Ablaufdiagramm

7.5.1. Ablaufdiagramm Parser

Zugunsten der Übersicht wurde die Logik der Expression in einem separaten Diagramm gespeichert.

Aus demselben Grund existieren bei der *if/else* & *while* Logik Methodenaufrufe von *Expression* und *nextStep.* Diese wurden an den betroffenen Stellen erwähnt, aber nicht eingezeichnet.




Start Expression
Expression
ja
Item = Operator?
ja
Operatorzeichen abspeichern
nein
Rechter Parameter abgefüllt?
ja
X-Wert abspeichern
Nächstes Zeichen = ")"/":" ?
nein
nein
Y-Wert abspeichern
Ende

8. Assembler

8.1. MIPS

MIPS bedeutet „Microprocessor without Interlocked Pipeline Stages". Es handelt sich dabei um eine Prozessorfamilie, die auf der RISC-Architektur basieren. Die Prozessoren werden vor allem in Embedded-Systems verwendet. Zu den populärsten Produkten mit solchen Prozessoren gehören die Play Station 2, als auch die PSP.
MIPS verfügt über 32 32-Bit Register. Angeprochen werden die Regiser jeweils mit $Registername, als zum Beispiel $a0.[3]

Registername	Registernummer	Nutzung	Bemerkung
zero	0	Enthält den Wert 0	Fix
at	1	Temporäres Assemblerregister	Nur durch den Assembler nutzbar
v0, v1	2,3	Funktionsergebnisse	
a0 -a3	4-7	Argumente 1-4 für Prozeduraufrufe	
t0 - t7	8-15	Temporäre Variablen 1-8	Können von Prozeduren verändert werden
s0 - s7	16-23	Langlebige Variablen 1-8	Können von Prozeduren nicht verändert werden
t8, t9	24,25	Temporäre Variablen 0 und 10	Können von Prozeduren verändert werden
k0, k1	26,27	Kernel-Register 1 und 2	Reserviert für das Betriebssystem
gp	28	Zeiger auf Datensegment	
sp	29	Stackpointer	Zeigt auf das erste freie Element des Stacks
fp	30	Framepointer	Zeiger auf den Prozedurrahmen
ra	31	Return address	Enthält nach dem Aufruf eines unbedingten Sprungs, die Rücksprungadresse

8.2. SPIM

Beim SPIM handelt es sich um ein MIPS Simulator. Das Programm simuliert R2000 und R3000 Prozessoren. Das Arbeiten mit einem solchen Simulator hat zwei entscheidende Vorteile. Bei einem Fehler, ist es ein Leichtes, den Simulator zurückzusetzen. Dies ist deutlich einfacher, als den Computer neu starten zu müssen. Der andere Vorteil ist, dass das Verhalten des Simulators auf Windows, Unix als auch Mac-Betriebssystemen gleich ist. Würde Assemblercode direkt für den Rechner entwickelt werden, würde dieser nur auf einem Betriebssystem ausführbar sein.[4]

8.3. Verwendete Assemblerbefehle[5]

Befehl	Erklärung
add dest, src1, src2	Addiert das Register src1 mit src2 und speichert das Resultat im Register dest
mult dest, src1, src2	Multipliziert das Register src1 mit src2 und speichert das

[3] https://www.scss.tcd.ie/~waldroj/itral/spim_ref.html, 12. Mai 2013
[4] http://www.inf.fu-berlin.de/lehre/SS00/19502-V/spimdoku.pdf, 12. Mai 2013
[5] https://www.scss.tcd.ie/~waldroj/itral/spim_ref.html, 12. Mai 2013

	Resultat im Register dest
div dest, src1, src2	Dividiert das Register src1 durch src2 und speichert das Resultat im Register dest
neg dest, src1	Negiert den Wert von src1 und speichert ihn im Register dest (wird für die Subtraktion verwendet)
seq dest, src1, src2	Setzt das Register dest auf 1, wenn der Wert des Registers src1 gleich dem von src2 ist. Sonst 0
sgt dest, src1, src2	Speichert den Wahrheitswert (0/1) in dest, des Vergleiches src1 >src2
sge dest, src1, src2	Speichert den Wahrheitswert (0/1) in dest, des Vergleiches src1 >=src2
sle dest, src1, src2	Speichert den Wahrheitswert (0/1) in dest, des Vergleiches src1 <=src2
slt dest, src1, src2	Speichert den Wahrheitswert (0/1) in dest, des Vergleiches src1 <src2
move dest, src1	Verschiebt den Wert aus src1 nach dest
la dest, address	Ladet die Adresse address ins Register dest
lw dest, address	Ladet das Wort, welches an der Adresse address abgespeichert ist ins Register dest
beq src1, src2, label	Bedingter Sprung zur Position label, wenn Register src1 gleich dem Wert src2 ist.
beqz src1, label	Bedingter Sprung zur Position label, wenn Register src1 0 ist
jal label	Unbedingter Sprung zur Position label
jr src1	Sprung zur Adresse, die im Register src1 liegt (häufig mit dem Register $ra verwendet).
li $v0, 4 syscall	Gibt den String, dessen Adresse im Register a0 liegt aus
li $v0, 1 syscall	Gibt die Zahl, die im Register a0 abgespeichert ist, aus
li $v0, 8 syscall	Liest einen Eingabe-String aus der Konsole in das Register $v0
li $v0, 5 syscall	Liest eine Zahleneingabe aus der Konsole in das Register $v0

9. Fazit

Anhand des empfohlenen Buches „Grundlagen und Techniken des Compilerbaus" von Niklaus Wirth, war es mir möglich, in vernünftiger Zeit einen Einblick in das Thema zu erhalten. Das Buch ist aufgrund des guten Aufbaus sehr empfehlenswert. Auch Personen ohne Vorwissen werden gut in die Thematik eingeführt.
Etwas gestört hat mich, dass die Codebeispiele mit Oberon-0 geschrieben wurden, was meines Erachtens keine gängige Programmiersprache ist. Jemand ohne Programmiererfahrung wird sich mit grösster Wahrscheinlichkeit nicht zurecht finden. Zum Glück habe ich mich nach gewissen Anlaufschwierigkeiten aber soweit zurechtgefunden, dass ich den Kontext der einzelnen Prozeduren verstehen konnte.
Die grössten Schwierigkeiten bereiteten mir aber die ersten Schritte in der Assemblerprogrammierung. Die erste Hürde stellte schon die Suche nach einem geeigneten RISC-Simulator dar. Nach einigem Umherirren im Web, stiess ich auf die Seite https://www.scss.tcd.ie/John.Waldron/itral/cahome.html. Es war die erste Page, die ich fand, welche neben einem schlanken Simulator auch gleich gut dokumentierte Code-Beispiele anbot.
Besonders gefreut habe ich mich auf die Implementierung. Bei diesem Projektabschnitt hatte ich auch keinerlei Startschwierigkeiten. Etwas aufwändiger stellte sich das Testen heraus. Bei jedem neuen Versuch, musste ein Dokument eingelesen werden. Der daraus resultierte Assembler-Code musste dann von Hand in PCSpim geladen und ausgeführt werden. Auch die Fehlerbehebung war zumeist nicht ganz einfach. Als erstes musste der Fehler im Assembler-Code gefunden und die Ursache dafür erkannt werden. Im zweiten Schritt musste die Stelle im Java-Code ermittelt werden, die den Fehlerhaften Assembler-Code erstellt hat.
Ich denke diese Arbeit war ein guter Moment, um einmal hinter die Kulissen der Programmierung zu werfen und einige Grundlagen des Compilerbaus zu erlernen. Weitere Projekte in diese Richtung werde ich aber nicht verfolgen. Des Weiteren erwies sich auch die Einarbeitung in die Assembler-Programmierung als interessant. Aber auch diese Gebiet ist neben der extrem schlechten Aufwand/Ertragssicht zu nervenaufreibend, als dass ich weiter damit Arbeiten werde.