Ostfalia
Hochschule für angewandte Wissenschaften

Fakultät Bau-Wasser-Boden

Pascal Ernst, Matrikelnummer 70302367

Realisierung eines Compilers für eine prozedurale Sprache

Abschlussarbeit zur Erlangung des Hochschulgrades
Bachelor of Science

im Studiengang Angewandte Informatik an der
Ostfalia Hochschule für angewandte Wissenschaften
– Hochschule Braunschweig/Wolfenbüttel

Betreuer: Prof. Dr. rer. nat. Albrecht Meißner, Jorin Kleimann M.Sc.

Inhaltsverzeichnis

Inhaltsverzeichnis I

Abbildungsverzeichnis III

Tabellenverzeichnis IV

Abkürzungsverzeichnis V

1 Einleitung und Motivation 1

I Compiler-Theorie 2

2 Der Compilerbau 3
2.1 Compilerhistorie . . . 3
2.2 Aufbau eines Compilers . . . 3
2.2.1 Der Parser . . . 4
2.2.2 Der Codegenerator . . . 6

3 Der Parser 7
3.1 Historisches / Details . . . 7
3.2 Top-Down Parser . . . 7
3.3 Bottom-Up Parser . . . 9
3.3.1 Shift-Reduce Parser . . . 9

4 Der Compiler im Zweipass-Verfahren 13

5 Die Zwischensprache im Compiler mit Zweipassverfahren 15
5.0.1 Drei-Adressen Code . . . 15

6 Compiler-Backend mit RISC-V 18
6.0.1 RISC-V als Zielplattform . . . 18

II Umsetzung des Compilers 20

7 Planung des Compilers 21

8 Umsetzung des Parsers 23
8.1 Der Aufbau der Programmiersprache Slang . . . 23
8.1.1 Programme . . . 24

8.1.2 Funktionen . . . 24
8.1.3 Variablendeklarationen . . . 25
8.1.4 Ausdrücke . . . 27
8.1.5 Zuweisungen . . . 28
8.1.6 Die for-Schleife . . . 29
8.1.7 Die if-Bedingung . . . 30
8.1.8 Funktionsaufrufe . . . 31
8.1.9 Spezialfunktionen . . . 33
8.2 Parserdefinition von GNU Bison . . . 34

9 Umsetzung der Zwischensprache **35**
9.1 Die Symboltabelle . . . 35
9.2 Der Drei-Adressen Code . . . 36
9.2.1 Generierung des Drei-Adressen Codes . . . 37
9.2.2 Funktionsdeklaration und Funktionsaufruf . . . 41

10 Umsetzung des RISC-V Backends **44**
10.1 Register und Speicheradressen . . . 44
10.2 Zuweisung von Drei-Adressen Code zu RISC-V Befehlen . . . 45
10.2.1 Arithmetische Operationen . . . 45
10.2.2 Sprünge . . . 46
10.2.3 Vergleichsoperationen . . . 47
10.2.4 Funktionsaufrufe . . . 49
10.2.5 Funktionsdefinitionen . . . 50

III Auswertungen und Fazit **53**

11 Ergebnisse **54**
11.1 Aspekt der Generalität . . . 54
11.2 Das Slang Frontend . . . 55
11.3 Die Zwischensprache . . . 55
11.4 Das RISC-V Backend . . . 55
11.5 Fazit . . . 56

Literaturverzeichnis **57**

Anhang

Abbildungsverzeichnis

Abb. 1 Top-Down-Syntaxanalyse 5
Abb. 2 Bottom Up Syntax Parsung 5

Abb. 3 Beispiel Grammatikdefinition 7
Abb. 4 Das erste Token a wird gelesen, und die Regel A wird geöffnet 8
Abb. 5 Die weiteren Tokens werden bis zur Vervollständigung des Syntaxbaums gelesen 8
Abb. 6 Linksrekursive Grammatikdefinition 8
Abb. 7 Aufgelöste linksrekursive Grammatikdefinition 9
Abb. 8 Im Bottom-Up Verfahren wird der Syntaxbaum nach und nach von den Blättern aufgebaut 9

Abb. 9 Flexibler Zweipass-Compiler mit mehreren möglichen Frontends und Backends 13

Abb. 10 Aufbau des SLICC Compilers 21

Tabellenverzeichnis

Tab. 1 Ablauf des Shift-Reduce Parsers 10
Tab. 2 Shift-Reduce Parser Tabelle 11
Tab. 3 Stateübergänge des Shift-Reduce Parsers 11
Tab. 4 Zuweisung von Variablen in einer fiktiven Symboltabelle zu Registern . . . 44

Abkürzungsverzeichnis

BNF Backus-Naur-Form

EBNF Erweiterte Backus-Naur-Form

SLICC Simple Language to Instruction Compiler in C++

SLang Simple Language

RISC Reduced Instruction Set Computing

CISC Complex instruction set computer

TAC Three-Address Code

1 Einleitung und Motivation

Der Compiler ist ein essentieller Bestandteil der Informatik und eines der zentralen Werkzeuge von Programmierern. Der absolute Großteil moderner Software wird in menschenverständlichen, höheren Programmiersprachen formuliert und mittels eines Compilers oder Interpreters in maschinenlesbaren Code übersetzt.

Ziel dieser Arbeit ist es, einen simplen und verständlichen Compiler zu entwickeln, der eine eigens definierte, C- und Pascal-ähnliche Programmiersprache in Maschinencode der Reduced Instruction Set Computing (RISC)-Architektur kompiliert. Dabei wird der Compiler modular aufgebaut, um weitere Programmiersprachen lesen und Maschinencode weiterer Architekturen unterstützen zu können.
Mit der Implementierung des Compilers wird Verständnis für die Funktionsweise und den Aufbau eines Compilers geschaffen und praxisbezogen umgesetzt.

Teil I

Compiler-Theorie

2 Der Compilerbau

2.1 Compilerhistorie

Die ersten Computer wurden direkt in Maschinensprache programmiert. Mithilfe von handgeschriebenen Notationen und Tabellen haben die Programmierer die Übersicht über die Programme behalten.
Mit der Zeit wurden die Aufgaben, die die Computer erledigen sollten, immer komplexer und größer. Den Überblick über die binären Instruktionen zu behalten, wurde damit auch schwieriger. Statt direkt in Binärcode zu arbeiten, war es leichter, die Operationen in Symbole wie "LOAD" und "ADD" zu schreiben und zu lesen. Diese Symbole wurden dann in einem Zwischenschritt durch einen symbolbasierten Assembler in die computerverständliche Maschinensprache übersetzt.
Zusätzlich dazu veränderte sich die Technik; mit der Einführung der Magnettrommelspeicher zum Beispiel kamen Herausforderungen in der Optimierung des Speicherzugriffs auf. Je nach Ort der Daten konnte die Dauer, auf die Daten mit Direktzugriff zuzugreifen, um einen Faktor von 50 variieren [3]. Mithilfe eines optimierenden Assemblers wurden die Adresszuweisungen der Daten optimiert, sodass die Zugriffsdauer auf diese Daten verringert wurde.
Die Instruktionstypen, die eine damalige Prozessorarchitektur der Computer lesen und bearbeiten konnte, waren sehr limitiert. Oftmals waren mehrere kleine Instruktionen nötig, um Schritte im Programm darzustellen, die mit einem größeren, expliziteren Instruktionstyp zusammengefasst werden könnten. Diese fehlenden Instruktionstypen wurden anfangs mit einem Interpreter hinzugefügt; ein Programm, das die Übersetzung der programmierten, komplexeren Befehle in für die Prozessorarchitektur lesbare Befehle übernahm. Da ein Interpreter selbst auf dem Computer ausgeführt werden muss, führt dies zu deutlichem Mehraufwand in der Ausführung der Programme. Die ersten Compiler wurden geschrieben, um dieses Problem zu lösen. Sie übersetzten die vom Programmierer geschriebenen Befehlslisten in für die Prozessorarchitektur des Computers verständliche Instruktionen. Die Compiler übernahmen mit der Zeit weitere Aufgaben wie Speicherzugriffsoptimierungen und dynamische Speicherallokation. Verschiedene Programmiersprachen abstrahierten die Prozessorarchitektur weiter. Sie vereinfachten die Implementierung von Programmen in domänenspezifischen Problemen wie Handel und Forschung und verbreiteten sich später mit dem Aufkommen von Allzweck-Programmiersprachen auch darüber hinaus [3].

2.2 Aufbau eines Compilers

Heutzutage gibt es eine große Bandbreite an Programmen, die Compiler genannt werden. Der Compiler im klassischen Sinne übersetzt in Programmiersprachen geschriebenen Programmcode in Maschinencode. Die "Cross-Compiler" hingegen übersetzen von einer Programmiersprache

in eine andere Programmiersprache. Zusätzlich dazu gibt es Compiler (oder auch "compilerähnliche Werkzeuge"), die auf ganz anderen Datenstrukturen arbeiten und zum Beispiel Logdateien in Tabellen kompilieren. Im Rahmen dieser Arbeit wird der klassische Compiler behandelt, mit dem eine Programmiersprache zu einer Maschinensprache kompiliert wird.
Generell gibt es zwei Möglichkeiten, wie diese Compiler aufgebaut werden. In dem Einpass-Compiler wird der Programmcode, der Eingabesatz, Zeichen für Zeichen eingelesen, welche in Lexeme, lexikalische Gruppen, eingeordnet werden. Ein Lexem wird eingelesen, geparst und der entsprechende Maschinencode wird generiert. Dann fährt der Einpass-Compiler mit den nächsten Eingabezeichen fort, bis das Ende des Programmcodes erreicht ist.
Der Mehrpass-Compiler hingegen liest anfangs den gesamten Programmcode ein und parst ihn mithilfe von mehreren Programmteilen, wobei der letzte den Maschinencode ausgibt[6].
Diese Compiler haben generell 6 Phasen, in denen sie arbeiten.
Die erste, die lexikalische Analyse, unterteilt das Programm in Lexeme, also Einheiten wie Variablennamen oder Operationszeichen.
Als nächstes überprüft die Syntaxanalyse die Struktur des Programmcodes mithilfe von grammatikalischen Regeln. Sie generiert mit den Lexemen zusätzliche Daten und ordnet die resultierenden Tokens in eine Baumstruktur, den Syntaxbaum, ein.

2.2.1 Der Parser

Es gibt zwei grundlegende Arten von Parsern im Compilerbau, den Top-Down und den Bottom-Up Parser.
Der Top-Down Parser baut den Syntaxbaum aus der Wurzel heraus und bricht dann die Tokens immer weiter im Detail runter.
Für den Quellcode "*int myVar = x + 2;*" liest der Parser zuerst den ganzen Ausdruck ein und teilt diesen dann auf; nachdem er diesen Ausdruck in die einzelnen Tokens heruntergebrochen und ebenfalls geparst hat, wird geschlussfolgert, dass der gesamte Ausdruck eine Variablenzuweisung ist.

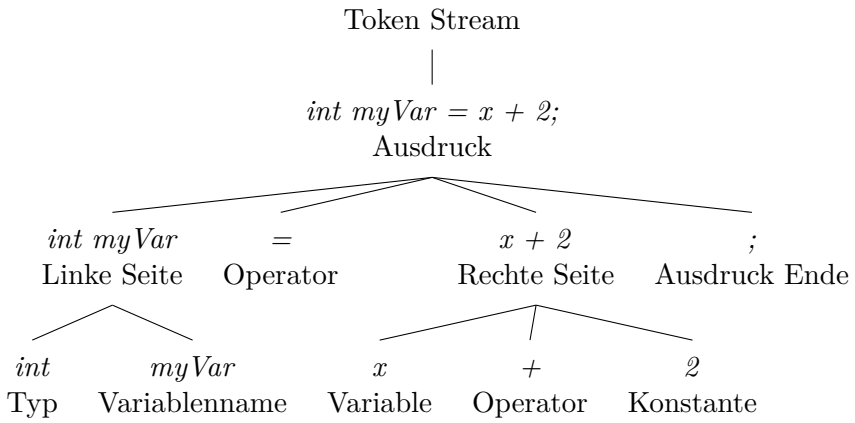


Abb. 1: Top-Down-Syntaxanalyse

Der Bottom-Up Parser fängt mit einzelnen Tokens aus dem Quellcode an und fügt diese zusammen[5]. Unser Ausdruck "*int myVar = x + 2;*" wird zuerst in viele generische Tokens geparst. So könnten anfangs "*myVar*" und "*x*" als Name identifiziert werden, deren konkrete Verwendung noch nicht bekannt ist. Erst mit dem Zusammenfügen und Parsen des gesamten Ausdrucks wird klar, dass "*myVar*" eine neue Variablendefinition ist und "*x*" eine existierende Variable, deren Wert benutzt wird um den Wert von "*myVar*" zu setzen.

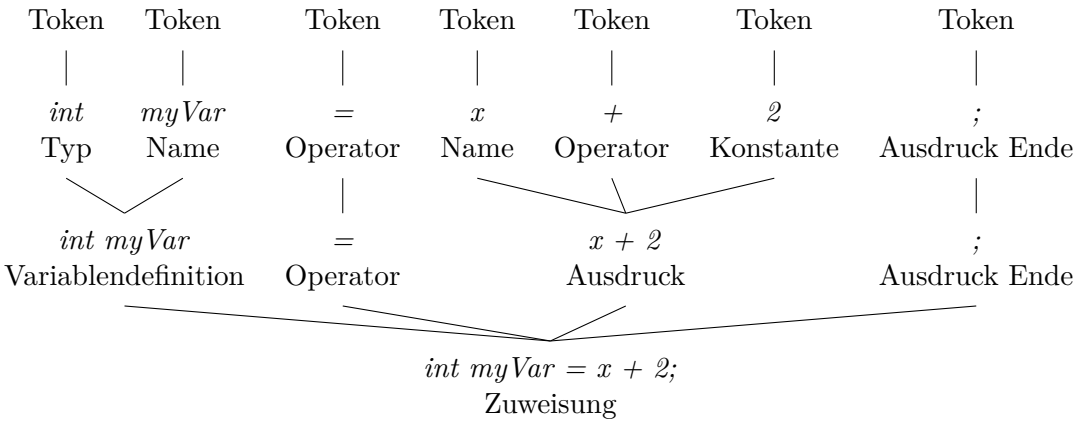


Abb. 2: Bottom Up Syntax Parsung

Nach dem Parsen der Tokens wird eine semantische Analyse auf dem Syntaxbaum ausgeführt. Hier werden unter anderem die Tokens als Symbole miteinander verknüpft und die Typen von Variablen überprüft. Wird zum Beispiel einer Variable von Typ "*int*" inkorrekterweise ein Wert vom Typ "*string*" zugewiesen, wird das in der semantischen Analyse als Fehler aufgezeigt. Nach diesen Schritten ist das grundlegende Parsen erfolgreich abgeschlossen.

Die Aufteilung des Quellprogramm-Einlesens in die lexikalische Analyse und Syntaxanalyse ist vor allem der Praktikalität der Implementierung komplexerer Programmiersprachen geschuldet. Ohne die Aufteilung müsste der Parser auch Zeichen für Formatierung und Lesbarkeit (z.B. Leerzeichen, Zeilenumbrüche, Kommentarstart und -ende) einlesen, was die Implementierung der Syntaxanalyse deutlich verkomplizieren würde.

2.2.2 Der Codegenerator

Einfache Compiler generieren aus den Symbolen direkt Ausgaben wie Maschinencode mit dem Codegenerator. In modernen Compilern werden vor diesem abschließenden Schritt noch Optimierungen vorgenommen, die den Maschinencode in Nutzung von Ressourcen wie Arbeitsspeicher, Effizienz von Befehlen in der Zielprozessorarchitektur und Größe der Binärdatei verbessern.

Der Codegenerator hat hauptsächlich die folgenden Aufgaben zu erfüllen [6]:
Zum einen müssen die geparsten Ausdrücke in für die angezielte Prozessorarchitektur lesbaren Maschinencode generiert und als Maschinenprogramm in einer Datei gespeichert werden. Weiterhin müssen Konstrukte wie Schleifen und Verzweigungen in Sprungbefehle übersetzt werden. Der Codegenerator muss sich auch um lokale Variablen von Methoden kümmern, die im richtigen Kontext initialisiert und dann wieder freigegeben werden müssen.
Während der Generierung ist auch Fehlerbehandlung wichtig, die verhindert, dass ein unlesbares oder inkorrektes Maschinenprogramm erzeugt wird.

Der Codegenerator ist stark von der angezielten Prozessorarchitektur und ihren Instruktionstypen abhängig. Die *Reduced Instruction Set Computing* (RISC) Architektur mit wenigen einfachen Instruktionstypen benötigt wenige Spezialfallbehandlungen und ist damit relativ simpel in einem Codegenerator umzusetzen. Im Gegensatz dazu steht die Architektur *Complex instruction set computer* (CISC), zum Beispiel der x86 Architektur-Familie. Mit einer Unmenge an Instruktionstypen, deren Funktionsweise sich teilweise überschneiden oder mehrere kleine Instruktionstypen zur Optimierung der Ausführung gruppiert werden, ist ein vollständiger, korrekter und optimierter Codegenerator für eine dieser Prozessorarchitekturen extrem groß und komplex.

3 Der Parser

3.1 Historisches / Details

Einige Arten haben sich etabliert, um den Parser eines Compilers zu implementieren. Die häufigsten davon sind Varianten des Top-Down und Bottom-Up Parsers, bei denen die Baumstruktur des Syntaxbaums entweder aus der Wurzel der Syntaxdefinition nach "unten" hin oder den einzelnen Blättern nach "oben" hin gebaut wird.
So gibt es zum Beispiel den Rekursiver-Abstieg Parser, der eine der einfachsten Varianten eines Parsers ist und auch ohne Werkzeuge von Hand schreibbar ist [6]. Demgegenüber stehen eine große Auswahl an Werkzeugen wie Parsergeneratoren, die aus Grammatikdefinitionen den Code zum Parsen automatisch generieren. Diese haben den Vorteil dass sie schwer zu implementierende Parservarianten, z.B. Bottom-Up Parser, automatisch implementieren. Beispiele für solche Werkzeuge sind YACC und ANTLR, die aus einer Grammatikdefinition in einer Notation wie zum Beispiel BNF / *Erweiterte Backus-Naur-Form* (EBNF) den Parsercode generieren.

3.2 Top-Down Parser

Der Top-Down Parser baut den Syntaxbaum aus der Wurzel der Grammatikdefinition heraus und bricht dann die Tokens immer mehr im Detail runter [5].
Um dies zu verdeutlichen erstellen wir uns eine einfache Grammatikdefinition und wenden diese auf einen Eingabesatz an.

$$\begin{aligned} A &\rightarrow a\ R\ b \\ R &\rightarrow y \mid z \end{aligned}$$

Abb. 3: Beispiel Grammatikdefinition

In der Definition ist A die Wurzel der Grammatik, die mit dem Token a startet und mit b endet. Dazwischen ist eine weitere Grammatikregel R referenziert. Diese besteht aus einer Entweder-Oder Regel, hier entweder das Token y oder z.
A, R sind hier *Nichtterminalsymbole*, die nur in Zwischenschritten während des Parsens vorkommen. a, b, y, z sind *Terminalsymbole*, die nicht weiter heruntergebrochen werden.
Beispielshaft übergeben wir einem Parser in der Funktionsweise des rekursiven Abstiegs mit dieser Grammatik den Eingabesatz ayb.
Der Parser liest das erste Token a, das mit dem ersten Token der Regel A übereinstimmt.

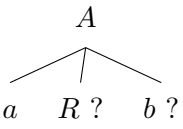


Abb. 4: Das erste Token a wird gelesen, und die Regel A wird geöffnet

Die Regel A wird geöffnet und das nächste Token y gelesen. da A jetzt nur R als mögliche Regel hat, wird R geöffnet und mit y aus dem Eingabesatz überprüft, der in $y|z$ enthalten ist. Zuletzt wird das Token b gelesen und damit die Wurzelregel A abgeschlossen. Da auch der Eingabesatz abgearbeitet ist, ist dieser gültig geparst worden und der Syntaxbaum vollständig [1, S.219].

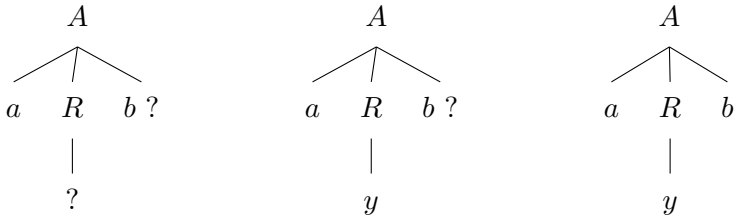


Abb. 5: Die weiteren Tokens werden bis zur Vervollständigung des Syntaxbaums gelesen

Diese einfache Art des Parsens hat aber einige Restriktionen, die durch die Grammatik eingehalten werden müssen.
So darf die Grammatik keine Linksrekursion enthalten, da der Parser sonst in eine Endlosschleife geraten könnte. Will man, wie in folgender Grammatikregel, A so erweitern, dass der Eingabesatz beliebig oft mit R und b forgeführt werden kann:

$$\begin{aligned} A &\rightarrow (A \mid a)\ R\ b \\ R &\rightarrow y \mid z \end{aligned}$$

Abb. 6: Linksrekursive Grammatikdefinition

Kann der Parser im rekursiven Abstieg die Regel A nicht mehr mit dem Eingabetoken lösen; Er muss dazu in A absteigen, aber damit rekursiv immer wieder A lösen.
Oftmals lassen sich solche Linksrekursionen durch Umformung der linksrekursiven Regel vermeiden. In diesem Fall kann A in zwei Regeln aufgeteilt werden, wie in Abbildung 7 dargestellt. END entspricht hier dem Ende des Eingabesatzes, also ob der Parser noch ein Token lesen konnte oder bereits alle Token eingelesen hat.
Dies ist jedoch nicht immer möglich. Eine mögliche Lösung ist die Grammatikdefinition zu verein-

fachen; Allerdings kann man auch eine andere Parservariante, zum Beispiel die eines Bottom-Up Parsers, benutzen, die mit Linksrekursion umgehen kann (aber ihre eigenen Vor- und Nachteile besitzt).

$$\begin{array}{lcl} A & \rightarrow & a\ B \\ B & \rightarrow & R\ b\ (END \mid B) \\ R & \rightarrow & y \mid z \end{array}$$

Abb. 7: Aufgelöste linksrekursive Grammatikdefinition

3.3 Bottom-Up Parser

Der Bottom-Up Parser baut den Syntax Baum von den Blättern der Grammatikdefinition bis zur Mit der Grammatikdefinition aus Abbildung 3 und dem Eingabesatz *ayb* liest der Parser das erste Token a ein, das noch nicht zu einer Regel zugeordnet werden kann. Der Parser speichert dieses Token zwischen und liest das nächste Token y ein. Dieses kann direkt der Regel R zugeordnet werden. Mit dem letzten Token b kann der Parser dann aus dem existierenden Teil des Baumes und dieses Token in die Regel A zusammenfassen.

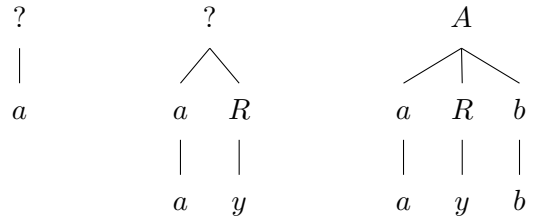


Abb. 8: Im Bottom-Up Verfahren wird der Syntaxbaum nach und nach von den Blättern aufgebaut

Der Bottom-Up Parser *reduziert* hier mithilfe der Tokens die Grammatikregeln zu einen Syntaxbaum, im Gegensatz zu dem Top-Down Parser, der mithilfe der Tokens den Syntaxbaum aus den Grammatikregeln *ableitet*.

3.3.1 Shift-Reduce Parser

Eine Variante des Bottom-Up Parsers ist der "Shift-Reduce" Parser, der die Tokens in einem Zwischenspeicher speichert und die Grammatikregeln aus den gelesenen Tokens und dem Zwischenspeicher reduziert. Im Gegensatz zur Top-Down Parser Familie sind dessen Grammatikdefinitionen weniger beschränkt, er behandelt zum Beispiel linksrekursive Grammatiken ohne

Probleme. Allerdings ist es schwieriger, die Semantikanalyse in den Parser zu integrieren, und die Implementation des Parsers selber ist wesentlich komplexer. So benötigt der Shift-Reduce Parser eine Parsertabelle. Ausserdem muss der Parser vorher geparste Token und daraus reduzierte Regeln zwischenspeichern, um diese im späteren Verlauf zu weiteren Teilen des Syntaxbaumes kombinieren zu können. Deswegen werden reale Bottom-Up Parser meistens von Parsergeneratoren aus Grammatikdefinitionen wie der *Backus-Naur-Form* (BNF) generiert [6].

Der Shift-Reduce Parser wird, wie viele Parser von kontextfreien Grammatiken, als Kellerautomat implementiert. Er speichert die Tokens, die er einliest, in einem Keller, und reduziert sie während des parsens aus der Grammatikdefinition zu dem Syntaxbaum. Ein Shift-Reduce Parser arbeitet grundlegend mit vier Operationen während des Einlesens der Tokens [6]:

- **shift**: Das gelesene Token wird vom Eingabesatz in den Keller verschoben.
- **reduce**: Tokens im Keller werden zu Nichtterminalsymbolen reduziert
- **accept**: Der Eingabesatz ist vollständig gelesen und der Syntaxbaum vollständig aufgebaut.
- **error**: Der Parser kann den Eingabesatz nicht weiter auf die Grammatikdefinition anwenden, der Eingabesatz ist fehlerhaft

Als Beispiel wird wieder der Eingabesatz $a\ y\ b\ \#$ (wobei # für das Ende des Eingabesatzes steht) mit der Grammatikdefinition aus Abbildung 3 geparst.

Keller	Eingabe	Aktion
	$\boldsymbol{a}\ y\ b\ \#$	$shift$
a	$\boldsymbol{y}\ b\ \#$	$shift$
$a\ y$	$b\ \#$	$reduce\ y \rightarrow \mathrm{R}$
$a\ R$	$\boldsymbol{b}\ \#$	$shift$
$a\ R\ b$	$\#$	$reduce\ a\ R\ b \rightarrow \mathrm{A}$
A	$\boldsymbol{\#}$	$accept$

Tab. 1: Ablauf des Shift-Reduce Parsers

In der Tabelle 1 sind die einzelnen Schritte des Parsers aufgeführt. Zuerst wird das Token a gelesen und in den Keller verschoben ($shift$). Da aus diesem Token kein Nichtterminalsymbol reduziert werden kann, wird das nächste Token y gelesen und ebenfalls gekellert. y hingegen kann in der Grammatikdefinition als R reduziert werden, was in Schritt 3 mit $reduce$ durchgeführt wird. a bleibt weiterhin unverändert im Keller, nur die Tokens zur Regel R werden mit dem Nichtterminalsymbol R im Keller ersetzt. Mit dem Lesen von Token b in Schritt 4 können wir alle Tokens im Keller mit der Regel für A in Schritt 5 reduzieren. Der Parser ist dann am

Ende des Eingabesatzes angekommen und konnte die Tokens vollständig zu der Wurzel in der Grammatikdefinition reduzieren.

Eine reale Implementation des Kellerautomaten wird mithilfe von Zustände, in die sich der Parser befinden kann, gelöst. Die Übergänge zwischen den Zustände werden in einer Parsertabelle festgehalten:

Zustand	a	b	y	z	#	A	R
0	s1					s5	
1			s2	s2			s3
2		rR					
3		s4					
4					rA		
5					acc		

Tab. 2: Shift-Reduce Parser Tabelle

Die Zustände werden im Keller gespeichert und ähnlich wie in der Tabelle 1 bei Reduzierungen im Keller entfernt. Die Kombination aus Zustand und Token gibt an, welche Aktion der Parser durchführen soll.
Mit sx wird das jetzige Zeichen geshifted und der Zustand x an den Keller angehängt.
rX reduziert das Nichtterminalsymbol X und entfernt die Zustände der Token von X, die am Ende des Kellers stehen. X wird dann benutzt, um den nächsten Stateübergang herbeizuführen.
acc beendet den Parser, wenn das Ende des Eingabesatzes (#) erreicht wird, erfolgreich.
Leere Tabellenfelder sind Fehlerzustände, bei denen der Eingabesatz ungültig ist.

Angewendet auf unser Beispiel $a\ y\ b\ \#$ ergibt sich folgender Ablauf:

Keller		Eingabe	Aktion
0		$\boldsymbol{a}\ \ y\ \ b\ \ \#$	$s1$
0 **1**		$\boldsymbol{y}\ \ b\ \ \#$	$s2$
0 1 **2**		$\boldsymbol{b}\ \ \#$	rR
0 1	**R**	$b\ \ \#$	$s3$
0 1 **3**		$\boldsymbol{b}\ \ \#$	$s4$
0 1 3 **4**		$\boldsymbol{\#}$	rA
0	**A**	$\#$	$s5$
0 **5**		$\boldsymbol{\#}$	acc

Tab. 3: Stateübergänge des Shift-Reduce Parsers

In der Parsertabelle ist gut zu erkennen, dass jeder Zustand für ein Token maximal einen Übergang hat. Das ist nur bei simplen Grammatiken der Fall. Man bezeichnet diese Grammatiken

als $LR(0)$ Grammatik. Das LR rührt vom Bottom-Up Parser, der von *L*inks nach *R*echts parst und deswegen auch LR-Parser genannt wird. Mit der 0 wird angegeben, dass der Parser kein "Vorgriffssymbol" benötigt, also nicht mehr als ein Token aus dem Eingabesatz im Voraus lesen muss, um den Syntaxbaum abzuleiten.
Komplexere Grammatiken werden $LR(n)$ genannt, bei der n die Anzahl der Vorgriffssymbole angibt, die ein Parser zum erfolgreichen Parsen der Grammatik benötigt. Diese werden nochmals in eine Subkatetegorie $LALR(n)$ eingeteilt, bei denen unter kleinen Einschränkungen in der Grammatikdefinition die Menge an Zustände in der Parsertabelle und somit der Kellerautomat deutlich vereinfacht werden kann.
Weit verbreitete Parsergeneratoren wie "YACC" und "GNU Bison" generieren aus diesen $LALR(1)$ Grammatiken den Bottom-Up Parser. Sie nutzen Vorteile wie die vereinfachte Fehlerbehandlungen durch die Parsertabelle und dem Fakt das $LALR(1)$ Grammatiken mächtiger sind als $LL(1)$ Grammatiken, die für Top-Down Parser verwendet werden.

4 Der Compiler im Zweipass-Verfahren

Historisch wurde der Mehrpass-Compiler dem Einpass-Compiler vorgezogen, da das Unterteilen der Einzelschritte in mehrere Programmteilen die Nutzung von Arbeitsspeicher verringerte und das Compilieren von komplexen Programmiersprachen ermöglichte. Mit den Verbesserungen an Hard- und Software ist beides kein Problem mehr. Deswegen werden dem Mehrpass-Compiler entweder der Einpass-Compiler oder eine spezielle Variante des Mehrpass-Compilers, der Zweipass-Compiler, heutzutage bevorzugt [6].
Im Zweipass-Compiler werden die Phasen des Compilers in zwei Programmteile gruppiert. Der erste Teil, das Frontend des Compilers, überprüft den Programmcode auf Syntax und Semantik; Aus diesen Informationen wird eine Zwischensprache generiert, die den Programmcode in einer internen Darstellung abbildet. Der zweite Teil, das Backend, liest die Zwischensprache ein und generiert daraus Maschinencode.

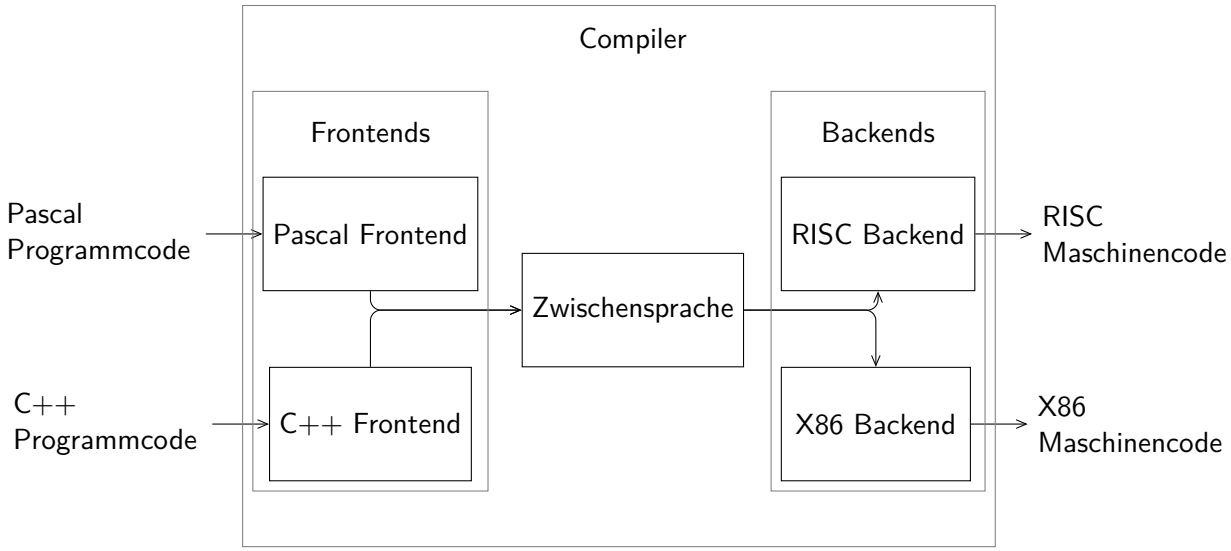


Abb. 9: Flexibler Zweipass-Compiler mit mehreren möglichen Frontends und Backends

Dieser Aufbau bietet einige Vorteile gegenüber dem Einpass-Compiler:
Zum einen können bei gleicher Zwischensprache mehrere Varianten von Frontends und Backends gebaut und benutzt werden; So kann man zum Beispiel eine andere Variante des Backends schreiben, um eine Programmiersprache in zwei unterschiedliche Prozessorarchitekturen zu übersetzen, ohne das Parsen nochmals zu schreiben. Das Gleiche gilt auch für die einzulesenden Programmiersprachen, die mit unterschiedlichen Frontends geparst und mit dem gleichen Backend zu Maschinencode verarbeitet werden.
Zum anderen ist die Zwischensprache ein guter Punkt, um Optimierungen an der Struktur, Speichernutzung und so weiter vorzunehmen. Diese generellen Optimierungen können dann noch von

den Backends um Optimierungen, die für die Prozessorarchitektur spezifisch sind, erweitert werden.
Da die Optimierung von Programmcode ein wichtiger Teil moderner Compilern ist, sind diese überwiegend Zweipass-Compiler [6].

Der in dieser Arbeit behandelte Compiler Simple Language to Instruction Compiler in C++ (SLICC) verwendet das Zweipass-Verfahren, um mehrere Ausgaben zu unterstützen und zukünftige Optimierungen zu ermöglichen.

5 Die Zwischensprache im Compiler mit Zweipassverfahren

Die Zwischensprache trennt das Compilerfrontend, das den Quelltext einliest und parst, von dem Compilerbackend, welches den Maschinencode generiert. Eine gute Zwischensprache abstrahiert die Details des Quelltextes und des Maschinencodes der Zielarchitektur, sodass die Frontends und Backends nicht voneinander abhängig sind. Damit kann man ein beliebiges Frontend mit einem beliebigen Backend kombinieren, und muss die jeweilige Komponente nur einmal schreiben.
Das Aussehen der Zwischensprache kann viele Varianten annehmen. So ist zum Beispiel ein Syntaxbaum, der beim Parsen des Quelltextes aufgebaut wird, bereits eine mögliche Zwischensprache. Sie kann aber auch eher Maschinencode-nah sein und bereits auf einzelne Operationen und Register verweisen. Für Optimierungen des Kompilates kann es sogar sinnvoll sein, mehrere Zwischensprachen zu verwenden, die aufeinander aufbauen und immer mehr Details des Maschinencodes abbilden.
Es gilt hier einen passenden Kompromiss zu finden, der die Optimierungsmöglichkeiten nicht einschränkt, aber auch nicht zu komplex ist.

5.0.1 Drei-Adressen Code

Eine Repräsentationsmöglichkeit von Operationen in einer Zwischensprache ist der "Drei-Adressen Code" (en. "Three-Address Code"). Hier werden namensgebend drei Adressen in einer Operation mit genau einem Operator verwendet[1]. Komplexere Operationen werden dadurch in mehrere dieser Operationen aufgeteilt, die dann leichter zu Maschinencode übersetzt werden können.
Eine mathematische Operation wie `x = a + (y * z)` wird in zwei Drei-Adressen Code Operationen umgewandelt:

```
t1 = y * z
 x = a + t1
```

Da wir hier noch im Bereich der Zwischensprache sind, können die Adressen abstrakter dargestellt werden. Sie können eine Konstante oder eine Referenz auf eine Variable in der Symboltabelle sein. Hier ist es auch möglich, andere Typen, wie zwischenzeitlich generierte Optimierungen, einzufügen[1].

Drei-Adressen codes sind nicht nur auf mathematische Operationen beschränkt; Sie können viele andere Operationen wie Kontrollstrukturen oder Funktionsaufrufe abbilden. Ein bedingter Sprung kann mit einem Operator und drei Adressen abgebildet werden. Operator und zwei

Adressen werden für die Bedingung genutzt, die dritte Adresse zeigt auf den Ort, wohin gesprungen wird:

```
if  x < y  goto  T
```

Mit dem Drei-Adressen code als Grundeinheit kann die Zwischensprache damit bereits viele Funktionalitäten eines Programmes abbilden ohne große und komplexe Strukturen zu benötigen. Nehmen wir den Ausdruck eines nicht-trivialen Quelltextes

```
int  x = 0;
int  y = 1;
while  x < 10  {
  if  x / 2 == 0  {
    x = x + y;
  }
}
```

dann kann dieser in Drei-Adressen Code umgewandelt werden:

```
 1:  x = 0
 2:  y = 1
 3:  t1 = x < 10
 4:  if  t1  goto  6
 5:  goto  12
 6:  t2 = x / 2
 7:  t3 = t2 == 0
 8:  if  not t3  goto  11
 9:  t4 = x + y
10:  x = t4
11:  goto  3
12:  end
```

Hier fällt auf, dass diese Repräsentation nicht allzu weit entfernt von Assemblercode tatsächlicher Prozessorarchitekturen ist, aber noch unabhängig von Hardware-spezifischen Konzepten

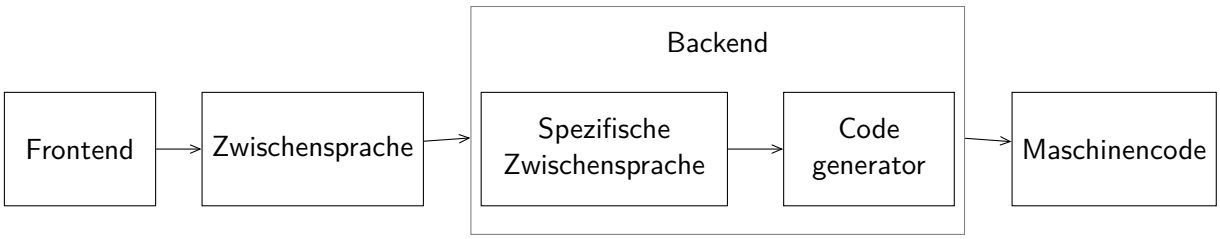
wie Register.
Dieser Code hat bereits Optimierungspotential, so können Zeile 7 und 8 kombiniert werden, indem der Vergleichsoperator von `t2 == 0` direkt negiert wird statt das Ergebnis `t3` zwischenzuspeichern und dann zu negieren.
Das Sprungziel von Zeile 8 ist eine Zeile die auch einen nicht bedingten Sprung initiiert, daher ist das Umschreiben von

`if not t3 goto 11` zu `if not t3 goto 3`

in Zeile 8 eine weitere Möglichkeit.

6 Compiler-Backend mit RISC-V

Das Backend im Zweiphasen-Compiler generiert aus der Zwischensprache und der Symboltabelle den Maschinencode, der am Ende von der Zielplattform ausgeführt wird. Die Zwischensprache kann bereits durch einen Optimierungsschritt gelaufen sein, aber diese sind nicht unbedingt auf den Maschinencode zugeschnitten, der generiert wird. Das Backend kann demnach seine eigenen Optimierungen, speziell für die angezielte Prozessorarchitektur, einbringen. Ein Zweiphasen-Compiler hat somit nicht zwangsweise nur Zwischensprachen zwischen dem Frontend und dem Backend; Backends können ihre eigenen Zwischensprachen zu Optimierungszwecken implementieren [1]:



Im Gegensatz zu den generischen Zwischensprachen eines Compilers werden Backendspezifische Zwischensprachen nicht zwischen Backends geteilt. Deswegen muss abgewägt werden, ob Optimierungen in der generischen Zwischensprache eingebaut werden können oder ob sie speziell für bestimmte Backends implementiert werden müssen und somit nur einzelne Backends von dieser Implementation profitieren.

6.0.1 RISC-V als Zielplattform

Die Zielarchitektur des in dieser Arbeit behandelten Compilers ist RISC. RISC ist ein Designprinzip für Prozessoren, das auf wenige, einfache und schnelle Instruktionen setzt. In den letzten Jahren hat die Architekturdefinition "RISC-V" immer mehr an Bedeutung gewonnen. Es ist ein offener Standard, der von der *RISC-V Foundation* entwickelt wird und eine verhältnismäßig simple Zielplattform bietet [**?**].

RISC-V besteht grundlegend aus einer Integer Befehlssatzarchitektur mit optionalen Erweiterungen dazu. Prozessoren müssen damit mindestens eine der beiden Varianten dieser Befehlssatzarchitektur, "RV32I" mit 32-bit Adressierung oder "RV64I" mit 64-bit Adressierung, implementieren. Eine spezielle Alternative für kleine Prozessoren, "RV32E", implementiert nur einen Teil der "RV32I", um Kosten dieser gering zu halten [2].
Eine interessante Erweiterung ist die "C" Erweiterung (zum Beispiel in "RV32C"). Sie erlaubt es, komprimierte 16-bit Befehle gemischt mit den Standard 32-bit Befehlen zu benutzen und dadurch die Programmgröße zu verringern. Nur Befehle, die auf eine bestimmte Art auf Register

zugreifen, können so komprimiert werden. Ein Compiler, welcher diese Erweiterung unterstützt, benötigt somit einen Optimierungsschritt, der für die RISC-V Zielarchitektur spezifisch ist. Diese Optimierung kann daher nur in dem speziellen Backend implementiert werden und nicht in der generischen Zwischensprache des Compilers.

Teil II

Umsetzung des Compilers

20

7 Planung des Compilers

Die Implementierung des Compilers soll folgende Ziele erfüllen

- Verständlichkeit und Inspizierbarkeit: Der Compiler soll inspizierbar und verständlich aufgebaut sein. Zusätzlich soll die Arbeitsweise des Compilers transparent sein, indem die einzelnen Teile des Compilers gut debugbar oder inspizierbar sind.
- Flexibles Backend: Der Compiler soll in der Lage sein, auf verschiedene Zielplattformen wie RISC Assembly und RISC Binary zu kompilieren.

Der in dieser Arbeit erstellte Compiler *Simple Language to Instruction Compiler in C++* (SLICC) wird hauptsächlich in C++ geschrieben, da die Werkzeuge wie die Parser-Generatoren, die benutzt werden, gut darauf aufbauen. Damit der Compiler selbst auf verschiedenen Plattformen möglichst interoperabel kompiliert werden kann, wird das robuste und weit verbreitete "CMake" zum Bauen der Binärdatei benutzt.
Da der Compiler ein Zweipass-Compiler ist, benötigen wir die Auftrennung des Compilers in Frontend und Backend mittels einer Zwischensprache. Die Zwischensprache wird in diesem Fall ein - im Theorie-Teil bereits erwähnter - Drei-Adressen Code sein, der die Operationen des Programmcodes in elementare Operationen mit drei Adressen aufteilt. Diese Zwischensprache trennt ein Frontend, das eine simple, Pascal-ähnliche Programmiersprache namens *Simple Language* (SLang) parst, von einem Backend, dass den Drei-Adressen Code in lesbaren RISC Assembly oder RISC Binary übersetzt und in eine Datei speichert.

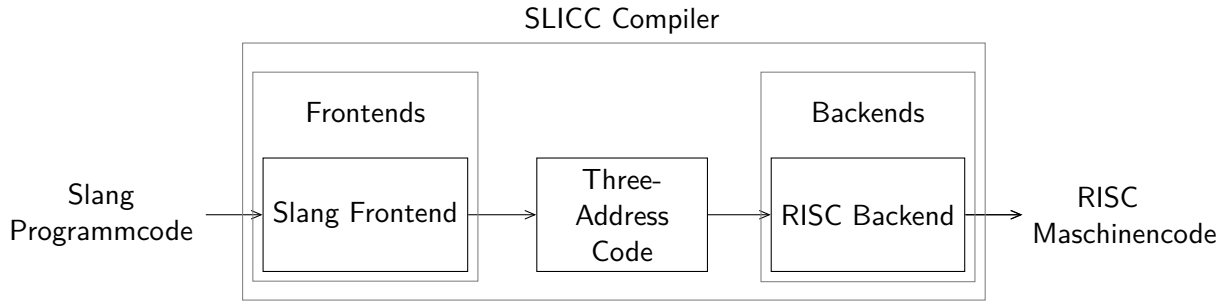
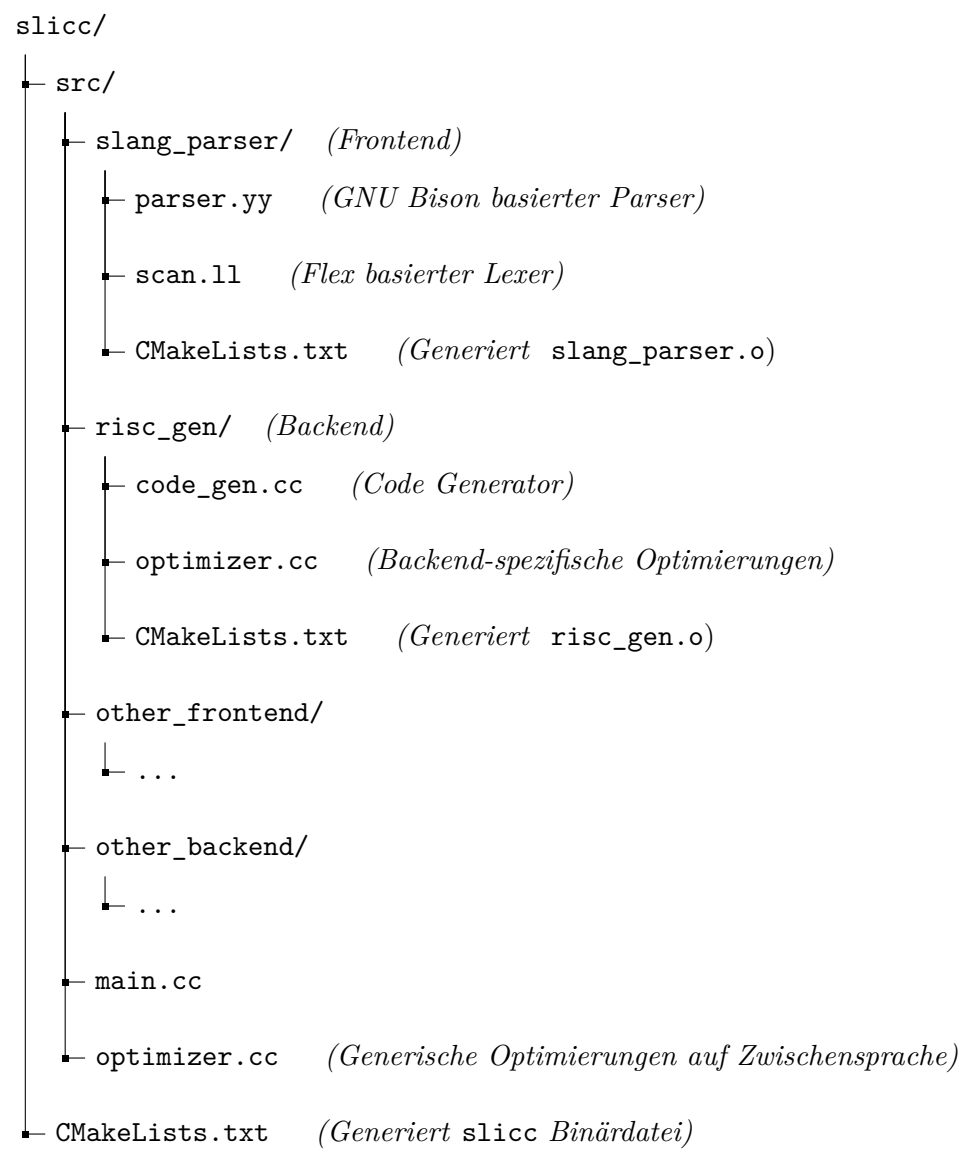


Abb. 10: Aufbau des SLICC Compilers

Da hier zunächst nur ein Frontend und ein Backend implementiert werden, muss die Entkopplung in der Implementierung des Compilers sichergestellt werden um die Flexibilität des Projektes zu wahren.
Um dies durchzusetzen, wird das Submodul-Feature von CMake benutzt, um das Frontend und Backend zu Bibliotheken zu kapseln, die dann vom Hauptprogramm, dem "Driver", eingebunden werden können. Als Ordnerstruktur wird folgendes gewählt:



Dadurch können dann in `main.cc` die Frontend- und Backend-Bibliotheken eingebunden und je nach Eingabeparameter bei der Ausführung ausgewählt werden.

8 Umsetzung des Parsers

Der angestrebte Parser ist ein Bottom-Up Parser generiert durch einen Parsergenerator. Die lexikalische Analyse wird von “Flex”, einer erweiterten Alternative zum klassischen “Lex”, übernommen. Flex parst eine Datei mit einer zu diesem Werkzeug spezifischen Syntax und schreibt den Lexer als automatisch generierten C- oder C++-Code in eine Datei. Die Schnittstelle dieses Lexers kann genutzt werden, um die Token während des Parsens weiterzuverarbeiten.
Flex bietet eine spezielle Schnittstelle, mit der Syntaxparsergeneratoren wie “Yacc” oder kompatible Derivate wie “GNU Bison” direkt integriert werden können, ohne in handgeschriebenen Programmcode verbunden werden zu müssen [9].
Bison bietet sich ebenfalls an, da es noch weiterentwickelt wird und das Yacc-kompatible Interface implementiert. Der Parsergenerator Bison implementiert interessante Möglichkeiten, wie zum Beispiel die Erweiterung des Parsers von *LR(1)*-Grammatiken zu *GLR* (Generalized LR) Grammatiken. Dadurch kann der Parser auch mit Shift-Reduce Konflikten umgehen, indem der Parser mehrere Parse-Stränge gleichzeitig verfolgt und bei Fehlern diese wieder schließt, bis nur der eine gültige Strang (bei korrekter Grammatikdefinition) weiterverfolgt wird [4].

8.1 Der Aufbau der Programmiersprache Slang

Die Programmiersprache Slang ist eine Pascal- und C-ähnliche Sprache, die in einer einzelnen Datei ein volles Programm repräsentiert. Sie unterstützt Funktionalitäten wie Variablendeklarationen, Zuweisungen, Kontrollstrukturen wie Schleifen und Verzweigungen sowie Funktionsaufrufe. Die Grundbausteine der Programmiersprache sind dabei die Definition von Programmen und Funktionen, in denen Anweisungsblöcke den Programmcode enthalten.

Listing 8.1: Grammatikdefinition Einsprungspunkt

```
<main> ::= <program> <wso> <func_def>* <wso>

<ws> ::= " " | "\t" | "\n"
<wso> ::= <ws>*
```

Wir definieren in unserer Grammatik zuerst eine Regel `<main>`, die mit einem `<program>` beginnt und dann eine beliebige Anzahl von Funktionsdefinitionen enthält.
Das Token `<wso>` steht für Formatierungszeichen, die in der Grammatik ignoriert werden. Wir koppeln Anweisungen und Blöcke mit Token wie Klammern (`{}`) und Semikolons `;` voneinander ab, damit die Formatierung des Programmcodes keine Rolle spielt, womit wir Leerzeichen, Tabs und Zeilenumbrüche ignorieren können.
In den nächsten Abschnitten gehen wir den Grammatikbaum von `<main>` durch, um die Gram-

matik von Slang komplett zu definieren.

8.1.1 Programme

Listing 8.2: Slang Beispiel Programmdeklaration

```
program my_calculations {

}
```

Ein “Programm” in Slang ist ein Anweisungsblock, der mit dem Token `program` gekennzeichnet wird. Ein Programm wird mit `<id>` benannt, was momentan aber nur der Lesbarkeit dient und keine praktische Funktion hat. Dies wird gefolgt von einem `<block>`, der die Anweisungen des Programms enthält.

<block> ::= <wso> "«wso> <block$_b$*ody* >< *wso* >< *wso* >< *id* >::=< *alphachar* > (< *alphachar* > | < *digit* > |_")∗ < *alphachar* >::= $[a-z]$ < *digit* >::= $[0-9]$

8.1.2 Funktionen

Listing 8.3: Slang Beispiel Funktionsdefinitionen

```
func void call_by_value(int val) {

}

func void call_by_reference(ref int val) {

}

func int call_with_return(int val) {

}
```

Funktionsdefinitionen in Slang werden mit dem Token `func` eingeleitet. Wenn zukünftig globalen Variablendeklarationen implementiert werden, macht es etwas leichter, diese grammatikalisch von den Funktionsdefinitionen zu unterscheiden.
Auf das `func` folgt der Typ des Rückgabewertes der Funktion, der entweder ein Datentyp wie `int` oder `void` sein kann, sowie eine `<id>`, die den Namen der Funktion angibt.
Danach folgt eine optionale Liste von Argumenten, die in Klammern stehen und durch Kommata

getrennt sind. Diese Argumente deklarieren Variablen, die in der Funktion verwendet werden können. Mit dem Token `ref` kann die Variable als Referenz in die Funktion übergeben werden, sodass die Funktion den Wert der Variable modifizieren kann, was sich dann auch auf den Wert im Kontext des Aufrufers auswirkt.

Listing 8.4: Grammatikdefinition Funktionsdefinitionen

```
<func_def> ::= "func" <wso> (<type> | "void") <wso> <id> <wso> "(" <wso>
    <func_args_def> <wso> ")" <wso> <block>

<func_args_def> ::= <func_arg_def>
                | <func_arg_def> <wso> "," <wso> <func_args_def>

<func_arg_def> ::= <type> <wso> <id>
                | "ref" <wso> <type> <wso> <id> <wso>
```

Damit wir die Typen und Variablen in den Funktionsdefinitionen definieren können, müssen wir auch ein paar Hilfsregeln definieren:

Listing 8.5: Grammatikdefinition Hilfsregeln

```
/* Typen und Variablen */
<type> ::= "int" | <intarray>

<intarray> ::= "int[" <intliteral> "]"

/* Zahlen */
<intliteral> ::= <digit>+
```

Damit sind alle Bausteine mit Blöcken definiert. Um die Programme und Funktionen sinnvoll benutzen zu können, brauchen wir nun die Regeln für die Anweisungen und Ausdrücke, die in diesen Blöcken stehen können.

8.1.3 Variablendeklarationen

Listing 8.6: Slang Beispiel Variablendeklarationen

```
program my_calculations {
    int i = 1;
    int foo;
    int[5] bar;
}
```

```
func void invalid(int val) {
  int x = val;
}
```

Die Blöcke der Programme und Funktionen beginnen mit einer Liste an Variablen, die deklariert werden. Slang unterstützt momentan zwei Datentypen: Integer `int` und eine Liste von Integer `int[x]`. Die Werte dieser Variablen werden standardmäßig auf 0 gesetzt. Ein Integer kann auch direkt mit einem Wert initialisiert werden, was aber bei den Arrays nicht implementiert ist.

Listing 8.7: Grammatikdefinition Variablendeklarationen

```
<block_body> ::= <variable_list> <wso> <statement_list>
              | <statement_list>

<variable_list> ::= <variable_declaration_stmt>
                 | <variable_declaration_stmt> <wso> <variable_list>

<variable_declaration_stmt> ::= (<variable_declaration> | <variable_initialization>)
    ";"

<variable_declaration> ::= <type> <wso> <id> <wso>

<variable_initialization> ::= <type> <wso> <id> <wso> "=" <wso> <intliteral> <wso>
```

Wir erlauben mit dieser Grammatik nur die Zuweisung von Integerwerten, nicht die Referenz von anderen Werten. In dem Beispielcode wird der lexikalische Parser in der Funktion `invalid` einen Fehler erzeugen, da Referenzen auf Variablen wie `val` nicht in der Deklarationsliste erlaubt sind.

Der lexikalische und syntaktische Parser kann mit dieser Grammatik aber nicht alle ungültigen Regeln abfangen; So erzeugt der folgende Code keinen Fehler, obwohl die Initalisierung von einem Array mit einer Integer nicht definiert ist:

Listing 8.8: Slang Beispiel: fehlerhafter Wert in der Initialisierung

```
program my_mistake {
    int[5] bar = 10;
}
```

Hier wird eine semantische Regel ausserhalb der Grammatikdefinition benötigt, die sicherstellt, dass die deklarierten Typen zu den initialen Werten passen.

8.1.4 Ausdrücke

Für Kontrollstrukturen oder mathematische Operationen benötigen wir Regeln für Ausdrücke, die an verschiedenen Stellen im Programmcode verwendet werden können.
Ein Ausdruck in Slang kann eine Integerkonstante, eine Variable, ein Funktionsaufruf, eine mathematische Operation oder eine vergleichende Operation sein.

Listing 8.9: Grammatikdefinition Ausdrücke

```
<expression> ::= <intliteral>
              | <id>
              | <func_call>
              | <expression> <wso> "+" <wso> <expression>
              | <expression> <wso> "-" <wso> <expression>
              | <expression> <wso> "*" <wso> <expression>
              | <expression> <wso> "/" <wso> <expression>
              | <expression> <wso> "%" <wso> <expression>
              | <compare_expression>
```

Da die Vergleichsoperation wahrscheinlich vermehrt in Kontrollstrukturen verwendet wird, ist diese Regel ausgelagert und vereinfacht.
Statt das generische `<expression> <comparison_operator> <expression>` zu erlauben, definiert die Regel `<compare_expression>` nur einfache Vergleiche zwischen zwei Werten, die Variablen oder Konstanten sein könnnen.
Damit wird die Komplexität von Vorbedingungen etwas ausgelagert.

Listing 8.10: Grammatikdefinition Vergleichsausdrücke

```
<comparison_operator> ::= "=="
                        | "!="
                        | "<"
                        | "<="
                        | ">"
                        | ">="

<compare_expression> ::= <id> <wso> <comparison_operator> <wso> <id>
                       | <id> <wso> <comparison_operator> <wso> <intliteral>
                       | <intliteral> <wso> <comparison_operator> <wso> <id>
                       | <intliteral> <wso> <comparison_operator> <wso> <intliteral>
```

8.1.5 Zuweisungen

Listing 8.11: Slang Beispiel Zuweisungen

```
program my_assignments {
    int i = 1;
    int foo;

    i = 5;
    foo = i + 5;
}
```

Die zweite Gruppe in einem Block sind die Anweisungen, die in Slang aus Variablenzuweisungen, Kontrollstrukturen, Schleifen, Funktionsaufrufen und `return` Anweisungen bestehen können.

In dem Block dürfen keine neuen Variablen deklariert werden; Dadurch wird sichergestellt dass die Symboltabelle alle in dem Block definierten Symbole enthält sobald diese referenziert werden können. Fälle wie Kontrollstruktur-lokale Variablendeklarationen, die nur in bestimmten Fällen deklariert sind und eine eigene Lebensdauer haben, müssen daher nicht durch den Compiler behandelt werden.

Listing 8.12: Grammatikdefinition Zuweisungen

```
<statement_list> ::= <statement>
                   | <statement> <wso> <statement_list>

<statement> ::= <statement_assignment> <wso> ";"
              | <if_statement>
              | <for_loop>
              | <func_call> <wso> ";"
              | "return" <wso> <expression> <wso> ";"

<statement_assignment> ::= <assigment_left> <wso> "=" <wso> <expression> <wso>

<assigment_left> ::= (<id>) | (<id> "[" <wso> (<intliteral> | <id>) <wso> "]")
```

Einfache Anweisungen in der `<statement_list>` sind durch Semikolons getrennt. Das `<if_statement>` und `<for_loop>` sind spezielle Anweisungen mit ihren eigenen Anweisungsblöcken, die durch geschweifte Klammern von den restlichen Anweisungen getrennt sind.
Als besondere Anweisung ist das `return <expression> ";"` definiert, das die Ausführung der Funktion oder des Programmes beendet und das Ergebnis des Ausdrucks zurückgibt.

8.1.6 Die for-Schleife

Listing 8.13: Slang Beispiel for-Schleife

```
program my_loop {
    int y;
    int i = 5;

    for (y < 5; y = y + 1) {
      bar[y] = x + y;
    }
    if (i == 5) {
      return 1;
    }

    return 0;
}
```

Slang unterstützt die `for`-Schleife, mit der ein Anweisungsblock wiederholt wird, solange eine Bedingung erfüllt ist. Im Gegensatz zu der for-Schleife in `C` gibt es keine Mögklichkeit, eine Variable im Kopf zu deklarieren.
Stattdessen muss diese Variable bereits in der Gruppe der Variablendeklarationen, die am Anfang eines Funktions- oder Programmblocks stehen, deklariert werden.

Listing 8.14: Grammatikdefinition for-Schleife

```
<control_block> ::= <wso> "{" <wso> <statement_list> <wso> "}" <wso>

<for_loop> ::= "for" <wso> "(" <wso> <compare_expression> ";" <wso>
    <statement_assignment> <wso> ")" <wso> <control_block>
```

Die `for`-Schleife beginnt mit dem Token `for`, gefolgt von einem Kopf, in dem ein Vergleichsausdruck und eine Zuweisung definiert werden. Auf den Kopf folgt ein Block mit den zu wiederholenden Anweisungen.

Da Variablendeklarationen nicht in Kontrollstrukturen erlaubt sind, führen wir eine weitere Regel `<control_block>` ein, die wie der `<block>` definiert ist, aber nur eine `statement_list` erlaubt.

8.1.7 Die if-Bedingung

Listing 8.15: Slang Beispiel if-Bedingung

```
program my_control {
    int i = 5;

    if (i == 5) {
      return 1;
    } else {
      return 0;
    }
}
```

Die `if`-Bedingung ist eine Kontrollstruktur, die einen Anweisungsblock ausführt, wenn eine Bedingung erfüllt ist. Optional folgt darauf ein `else`-Block, der ausgeführt wird, wenn die Bedingung nicht erfüllt ist.

Listing 8.16: Grammatikdefinition if-Bedingung

```
<if_statement> ::= "if" <wso> "(" <wso> <compare_expression> <wso> ")" <wso>
    <control_block>
                | "if" <wso> "(" <wso> <compare_expression> <wso> ")" <wso>
                    <control_block> <wso> "else" <wso> <control_block>
```

Ähnlich der `for`-Schleife beginnt die mit dem namensgebenden Token `if`, gefolgt von einer Bedingung in Klammern und einem Anweisungsblock. Auf diesem kann mit dem Token `else` und einem weiteren Anweisungsblock alternative Anweisungen ausgeführt werden.

8.1.8 Funktionsaufrufe

Listing 8.17: Slang Beispiel Funktionsaufrufe

```
program my_calls {
    int foo = 1;
    int y = 5;

    call_by_value(foo);
    call_by_reference(y);
    foo = call_with_return(y);
}

func void call_by_value(int val) {

}

func void call_by_reference(ref int val) {
  val = 2;
}

func int call_with_return(int val) {
  return val + 1;
}
```

Ein Funktionsaufruf in Slang besteht aus dem Funktionsnamen und einer Liste von Ausdrücken als Argumenten, die in Klammern stehen und durch Kommata getrennt sind.
Um den Rückgabewert einer Funktion zu speichern, kann der Funktionsaufruf auch als Ausdruck in einer Zuweisung verwendet werden.

Listing 8.18: Grammatikdefinition Funktionsaufrufe

```
<func_call> ::= <id> <wso> "(" <wso> <func_args> <wso> ")" <wso>

<func_args> ::= <expression>
             | <expression> <wso> "," <wso> <func_args>
```

Der Compiler überprüft mit semantischen Regeln dass die Anzahl der Argumente und deren Typen mit der refernzierten Funktionsdefinition übereinstimmen.
Der Wert eines Arguments, dass per Call-by-Value an eine Funktion übergeben wird, wird für diesen Ausführungsblock kopiert. Wird dieses Argument nochmals per Call-by-Reference an eine andere Funktion übergeben, wird der Wert nur in der ersten Funktion verändert, nicht in deren Aufruf:

Listing 8.19: Call-by-Value und Call-by-Reference

```
program my_calls {
    int y = 5;
    call_by_value(y);
    // y = 5
}

func void call_by_value(int val) {
  edit_by_reference(val);
  // val = 2
}

func void edit_by_reference(ref int val) {
  val = 2;
}
```

8.1.9 Spezialfunktionen

Listing 8.20: Slang Beispiel I/O Funktionen

```
program my_io {
  int y;
  read(y);
  write(y);
}
```

Für I/O bietet Slang zwei spezielle Funktionen, `read(x)` und `write(x)`, an, die in der Standardbibliothek implementiert sind.
`read` liest einen Wert von der Standardeingabe und speichert diesen in der übergebenen Variable, während `write` den Wert einer Variablen auf die Standardausgabe schreibt.

Da diese auf der Grammatik der Funktionsaufrufe basieren, sind diese nicht speziell in der EBNF Grammatikdefinition aufgelistet.
Stattdessen werden sie als Symbole in der Symboltabelle vorgeladen, sodass sie in der semantischen Analyse als Funktionen erkannt werden können und im Drei-Adress-Code referenziert werden können.

Damit ist die Grammatikdefinition und Auflistung der Funktionen für die Programmiersprache Slang komplett. Ein umfassenderes Beispiel für ein Slang-Programm ist in Anhang A zu finden. Die komplette EBNF Definition kann in Anhang B nachgeschlagen werden.

8.2 Parserdefinition von GNU Bison

Während einige Parsergeneratoren mit EBNF oder BNF Grammatiken arbeiten, arbeitet Bison mit einem eigenen Definitionsformat, welches von Yacc übernommen und erweitert wurde. Diese ist daran angelehnt, beim Parsen direkt C oder C++ Code auszuführen, der den einzelnen Grammatikregeln zugewiesen wurde.
Eine typische Regel in einer Bison `.y`-Datei definiert also nicht nur die Grammatikregel, sondern was mit den Werten, die aus dieser herausgelesen werden können, passieren soll:

Listing 8.21: Bison Beispiel

```
/* parser.yy */

...

%token TOK_PLUS "+"
%token TOK_MINUS "-"
%token <value> TOK_INTLITERAL

%%

expr:
  TOK_INTLITERAL { $$ = $1; }
  | expr TOK_PLUS expr { $$ = $1 + $3; }
  | expr TOK_MINUS expr { $$ = $1 - $3; }
  ;

%%
```

In diesem Beispiel wird ein einfacher Parser für mathematische Ausdrücke definiert, der die Addition oder Subtraktion zweier Integer berechnet. Hierbei ist der Ausdruck in den geschweiften Klammern Template-Code, der beim Parsen aufgerufen wird, um aus dem Parser Daten zur Weiterverarbeitung zu bekommen. Zum Beispiel wird hier der mathematische Ausdruck, der geparst wird, direkt ausgeführt. Dabei bezeichnen `$1` und `$3` Die Werte der geparsten Token an der Stelle 1 und 3. Beide lassen sich auf das Terminalsymbol `TOK_INTLITERAL` zurückführen, welches vom Lexer bereits in ein Integer übersetzt und als `<value>` reingegeben wurde. Das Ergebnis der Ausführung des mathematischen Asudrucks wird in `$$` gespeichert, das über eine Schnittstelle auslesbar ist.
Die Behandlung des Syntaxbaums wird somit von Bison komplett übernommen, der Implementierer muss nur noch den Wert, der an dieser Stelle des Baumes steht, definieren.

9 Umsetzung der Zwischensprache

Die Zwischensprache im Compiler besteht aus der Symboltabelle, die definierten Variablen und Funktionen speichert, und dem Drei-Adressen Code, der die Operationen auflistet.
Da die Datentypen dieser Konzepte die Schnittstelle zwischen Frontend und Backend darstellen, werden sie in dem globalen Namespace im Compiler definiert. Die Compiler-Frontends generieren bei der Analyse des Quelltextes die Daten, die die Symboltabelle und den Drei-Adressen Code repräsentieren, der dann an das Compiler-Backend weitergegeben und dort zu Maschinencode weiterverarbeitet wird.

9.1 Die Symboltabelle

Die Symboltabelle speichert alle Attribute, die zu einer Deklaration eines Symbols gehören. Als Datentyp verwendet der SLICC Compiler eine Liste `std::list` auf ein `struct`, welches alle potenziellen Attribute eines Symbols speichert.
Die Attribute sind:

- **name**: Der Name des Symbols, z.B. der Variablenname, Funktionsname oder Programmname
- **type**: Der Typ des Symbols aus einem enum, z.B. eine Variable vom Typ `SymbolType::INT`, eine Funktion vom Typ `SymbolType::FUNC` oder ein Funktionsargument vom Typ $\texttt{SymbolType::FUNC}_A$
  $Die Zeile, in der das Symbol definiert wurde$
- **col**: Das Zeichen in der Zeile, in der das Symbol definiert wurde
- **element_count**: Die Größe eines Arrays, wenn das Symbol ein Array ist
- **parent**: Der Name des Elternsymbols, z.B. der Funktionsname, in der das Symbol definiert wurde
- **func_arg_position**: Die Position des Arguments in der Funktionsdeklaration, wenn das Symbol ein Funktionsargument ist
- **is_ref**: Ein boolscher Wert, der angibt, ob das Funktionsargument ein Call-by-Reference Argument ist

Damit Einträge aus dem Drei-Adressen Code eindeutig Einträge in der Symboltabelle referenzieren können, wird die Kombination aus Name (`name`) und Elternsymbol (`parent`) als eindeutiger Schlüssel verwendet. Theoretisch kann die Symboltabelle durch das Elternsymbol mehrere Ebenen von Symbolabhängigkeiten speichern (`parent1 < parent2 < mySym`), aber in der Praxis sind die Frontends durch den eindeutigen Schlüssel auf eine Ebene beschränkt. Theoretisch könnten sie die Hierarchie manuell abbilden (z.B. durch `mySym.parent = "parent1:parent2"`),

was aber im Backend auch unterstützt werden müsste und diese miteinander koppeln würde.

9.2 Der Drei-Adressen Code

Der Drei-Adressen Code ist eine Liste `std::list` von mit `struct` repräsentierten Operationen, die in der Listen-Reihenfolge ausgeführt werden.

Listing 9.1: Drei-Adressen Code Eintrag

```
typedef struct _TacEntry {
  TacOperation op;
  TacArg* arg1;
  TacArg* arg2;
  char *res_ref;
} TacEntry;
```

Die Struct besteht aus dem Operationstyp `op`, dargestellt durch den enum `TacOperation`, den beiden Argumenten `arg1` und `arg2` sowie einer Referenz für das Ergebnis `result_ref`. Je nach Operationstyp werden nur bestimmte Teile der Argumente verwendet, um die Operation auszuführen:

- **ADD, SUB, MUL, DIV, MOD** arithmetische Operationen:
  `result_ref := arg1 op arg2`
- **LT, GT, LE, GE, EQ, NE** Vergleichsoperationen:
  `result_ref := arg1 op arg2`
  0 wird für `true` und 1 für `false` zurückgegeben
- **ASSIGN** Zuweisung eines Werts zu einem Symbol:
  `result_ref := arg1`
- **GOTO** Sprung zu einem Label:
  `goto result_ref`
- **LABEL** Sprungziel:
  `label result_ref`
- **COND_GOTO** Bedingter Sprung zu einem Label:
  `if arg1 == arg2 goto result_ref`
- **COND_NOT_GOTO** Bedingter Sprung zu einem Label:
  `if arg1 != arg2 goto result_ref`
  Diese Negation des `COND_GOTO` generiert für manche Kontrollstrukturen verständlicheren

Drei-Adressen Code als direkt `COND_GOTO` zu nutzen. Sonst müsste die die vorherige Vergleichsoperation, die bei diesen Kontrollstrukturen generiert wird, immer negiert werden.

- **CALL** Ruft eine Funktion auf, indem es zu diesem Label springt:
  `result_ref := arg1.var_ref`
- **RET** Beendet den ausgeführten Funktionsaufruf, gibt einen Wert zum **CALL** zurück wenn gesetzt:
  `return valueof result_ref`
- **REF** Verlinkt ein neues Symbol zu einem existierenden Symbol, sodass beide auf die gleiche Adresse zeigen:
  `result_ref := same_value_as arg1.var_ref`

Die beiden Argumente `arg1` und `arg2` werden durch ein `union` abgebildet, die entweder eine Referenz auf ein Symbol `sym_ref` oder einen Integer-Wert `int_val` speichert.

Listing 9.2: Drei-Adressen Code Argument

```
typedef union _TacArg {
  char *var_ref;
  int int_val;
} TacArg;
```

9.2.1 Generierung des Drei-Adressen Codes

Um die höheren Konzepte des Slang Quelltextes in den Drei-Adressen Code zu übersetzen, müssen die Frontends aus den Deklarationen und Statements des Quelltextes die passenden Operationen generieren.

Variablendeklaration

Listing 9.3: Beispielzuweisungen

```
int x = 5;
int y;
int[3] z;
```

Alle Variablen, die deklariert werden, benötigen nur eine Three-Address Code (TAC) Operation, die die Zuweisung eines Wertes zu einem Symbol darstellt. Da die Werte der `int` Variablen auf den angegebenen Wert initialisiert oder bei Deklaration auf 0 gesetzt werden, wird für jede Deklaration eine `ASSIGN` Operation generiert.

Bei der Deklaration eines Arrays wird jedem Element des Arrays der Wert 0 zugewiesen.

Listing 9.4: Drei-Adressen Code für Variablendeklaration

```
ASSIGN, arg1: {int_val: 5}, arg2: NULL, result_ref: "x"

ASSIGN, arg1: {int_val: 0}, arg2: NULL, result_ref: "y"

ASSIGN, arg1: {int_val: 0}, arg2: NULL, result_ref: "z[0]"
ASSIGN, arg1: {int_val: 0}, arg2: NULL, result_ref: "z[1]"
ASSIGN, arg1: {int_val: 0}, arg2: NULL, result_ref: "z[2]"
```

Zuweisungen

Listing 9.5: Beispielzuweisungen

```
x = 5;
y = x + 2 * 3;
z[1] = 3;
```

Zuweisungen funktionieren ähnlich wie Variablendeklarationen, aber anstatt den Wert `0` zu initialisieren, wird der Wert des Ausdrucks auf der rechten Seite der Zuweisung berechnet und zugewiesen.
Die Variable `y` hat eine nicht-triviale Berechnung, die in mehrere Operationen aufgeteilt wird. Slang definiert keine Priorität der Operationsreihenfolge, daher wird die Reihenfolge der Operationen durch die Reihenfolge der Argumente bestimmt.

Listing 9.6: Drei-Adressen Code für Zuweisungen

```
ASSIGN, arg1: {int_val: 5}, arg2: NULL, result_ref: "x"

ADD, arg1: {var_ref: "x"}, arg2: {int_val: 2}, result_ref: ":t1"
MUL, arg1: {var_ref: ":t1"}, arg2: {int_val: 3}, result_ref: "y"

ASSIGN, arg1: {int_val: 3}, arg2: NULL, result_ref: "z[1]"
```

Sobald Operationen in mehrere Operationen aufgeteilt werden, müssen temporäre Symbole Zwischenzustände speichern. Hier wird ein temporäres Symbol `:t1` verwendet, dass in die Symboltabelle eingetragen wird, um für das nächste `MUL` wieder benutzt werden zu können.

if-Bedingungen

Listing 9.7: Slang Vorlage if-Bedingung

```
if (<comp_expr>) {
  <body>
} else {
  <else_body>
}
```

Die Blöcke der `if`-Bedingungen werden mithilfe von `GOTO`s und `LABEL`s auseinandergehalten.

Listing 9.8: Drei-Adressen Code Vorlage für if-Bedingung

```
<comp_expr>, result_ref: ":t1"
COND_NOT_GOTO, arg1: {var_ref: ":t1"}, arg2: {int_val: 0}, result_ref: "ELSE_LABEL"

<body>

GOTO, arg1: NULL, arg2: NULL, result_ref: "ELSE_END_LABEL"
LABEL, arg1: NULL, arg2: NULL, result_ref: "ELSE_LABEL"

<else_body>

LABEL, arg1: NULL, arg2: NULL, result_ref: "ELSE_END_LABEL"
```

Für `if`-Bedingungen wird ein bedingter Sprung zu `ELSE_LABEL` generiert, der ausgeführt wird, wenn der Vergleich nicht `0` zurückliefert. Ist dies nicht der Fall, wird der `<body>` ausgeführt und am Ende des `<body>` mit einem `GOTO ELSE_END_LABEL` der `<else_body>` übersprungen.
Wird der bedingte Sprung ausgeführt, wird der `<else_body>` ausgeführt. Am Ende des `<else_body>` wird ein `ELSE_END_LABEL` eingefügt, damit bei Ausführung des `<body>` der `<else_body>` übersprungen werden kann.

Beispiel:

Listing 9.9: Beispiel if-Bedingung

```
if (x < 5) {
  y = 5;
}
```

Listing 9.10: Drei-Adressen Code für die if-Bedingung

```
LT, arg1: {var_ref: "x"}, arg2: {int_val: 5}, result_ref: ":t1"
COND_NOT_GOTO, arg1: {var_ref: ":t1"}, arg2: {int_val: 0}, result_ref: "L1"

ASSIGN, arg1: {int_val: 5}, arg2: NULL, result_ref: "y"

LABEL, arg1: NULL, arg2: NULL, result_ref: "L1"
```

for-Schleifen

Listing 9.11: Beispielschleife

```
for (<comp_expr>; <assign_expr>) {
  <body>
}
```

ie Vorlage für die For-Schleife mit den Platzhaltern aus Listing 9.11 hat die folgende Struktur:

Listing 9.12: Drei-Adressen Code für for-Schleifen

```
LABEL, arg1: NULL, arg2: NULL, result_ref: "START_LABEL"
<comp_expr>, result_ref: ":t1"
COND_NOT_GOTO, arg1: {var_ref: ":t1"}, arg2: {int_val: 0}, result_ref: "END_LABEL"

<body>
<assign_expr>

GOTO, arg1: NULL, arg2: NULL, result_ref: "START_LABEL"

LABEL, arg1: NULL, arg2: NULL, result_ref: "END_LABEL"
```

Die Bedingung der `for`-Schleife wird in einem bedingten Sprung abgebildet, der, erst wenn die Bedingung `<comp_expr>` nicht mehr erfüllt ist, zu dem Label `END_LABEL` springt, das die Schleife beendet. Passiert der bedingte Sprung nicht, wird zuerst der Block der Schleife ausgeführt und danach die Zuweisung `assign_expr` aus dem Schleifenkopf. Dann wird zurückgesprungen zum `START_LABEL` und die `<comp_expr>` sowie der bedingte Sprung erneut ausgeführt.

Zum Beispiel wird aus folgender Schleife

Listing 9.13: Beispielschleife

```
int i;
```

```
for (i < 5; i = i + 1) {
  x = x + 1;
}
```

der folgende Drei-Adressen Code generiert:

Listing 9.14: Drei-Adressen Code für for-Schleifen

```
ASSIGN, arg1: {int_val: 0}, arg2: NULL, result_ref: "i"

LABEL, arg1: NULL, arg2: NULL, result_ref: "L1"
LT, arg1: {var_ref: "i"}, arg2: {int_val: 5}, result_ref: ":t1"
COND_NOT_GOTO, arg1: {var_ref: ":t1"}, arg2: {int_val: 0}, result_ref: "L2"

ADD, arg1: {var_ref: "x"}, arg2: {int_val: 1}, result_ref: ":t2"
ASSIGN, arg1: {var_ref: ":t2"}, arg2: NULL, result_ref: "x"

ADD, arg1: {var_ref: "i"}, arg2: {int_val: 1}, result_ref: "i"
GOTO, arg1: NULL, arg2: NULL, result_ref: "L1"

LABEL, arg1: NULL, arg2: NULL, result_ref: "L2"
```

9.2.2 Funktionsdeklaration und Funktionsaufruf

Listing 9.15: Vorlage Funktionsdeklaration und Funktionsaufruf

```
myFunc(<args>);

func <type> myFunc(<arg_types>) {
  <func_body>

  return <type_val>
}
```

Funktionsdeklarationen erwarten, dass vordefinierte Funktionsargumente namens `:args_x` (mit `x` als Position des Arguments) in der Symboltabelle vorhanden sind. Bei einem Funktionsaufruf werden zuerst TACs generiert die diese `:args_x` auf die Symbole, die als Argumente in den Funktionsaufruf gegeben werden, setzen. Für Argumente, die nicht per Referenz übergeben werden, wird eine `ASSIGN` Operation generiert, die den Wert des Funktionsarguments in ein temporäres Symbol speichert, der dann weiter genutzt wird.
Für Argumente, die per Referenz übergeben werden, wird eine `REF` Operation generiert, die ein neues temporäres Symbol generiert, welches auf die gleiche Adresse wie das verlinkte Symbol

zeigt.
Dann wird mit einem `CALL` zum `<func_body>` gesprungen und dieser weiter ausgeführt.
Am Ende des Funktionsaufrufs wird ein `RET` generiert, das den Wert des `<type_val>` (wenn gegeben) zurückgibt.
Es kann auch vorher ein `RET` durch einen frühen `return` Befehl ausgeführt werden, dann wird die Ausführung des `<func_body>` abgebrochen und der Wert dieses `return` Befehls zurückgegeben.

Nehmen wir die folgende Funktionsdeklaration und den Funktionsaufruf als Beispiel:

Listing 9.16: Beispiel Funktionsdeklaration und Funktionsaufruf

```
int x = 5;
int y = 3;
int z = 0;

z = myFunc(x, y);

func int myFunc(int xa, ref int yb) {
  xa = 20;
  yb = 10;
  if (xa == 20) {
    return xa;
  }
}
```

Es wird zuerst eine Funktionsdefinition generiert, die den Wert des Arguments `xa` dupliziert und die Adresse des Arguments `yb` referenziert.

Listing 9.17: Drei-Adressen Code Funktionsdeklaration

```
GOTO, arg1: NULL, arg2: NULL, result_ref: "LABEL_MY_FUNC_END"

LABEL, arg1: NULL, arg2: NULL, result_ref: "LABEL_MY_FUNC"
ASSIGN, arg1: {var_ref: ":args_0"}, arg2: NULL, result_ref: ":t1"
REF, arg1: {var_ref: ":args_1"}, arg2: NULL, result_ref: ":t2"

ASSIGN, arg1: {int_val: 20}, arg2: NULL, result_ref: ":t1"
ASSIGN, arg1: {int_val: 10}, arg2: NULL, result_ref: ":t2"

// if-Bedingung
EQ, arg1: {var_ref: ":t1"}, arg2: {int_val: 20}, result_ref: ":t3"
COND_NOT_GOTO, arg1: {var_ref: ":t3"}, arg2: {int_val: 0}, result_ref: "L1"
RET, arg1: {var_ref: ":t1"}, arg2: NULL, result_ref: NULL
```

```
LABEL, arg1: NULL, arg2: NULL, result_ref: "L1"

// Standard Return falls nichts zurückgegeben wird
RET, arg1: {var_ref: 0}, arg2: NULL, result_ref: NULL

LABEL, arg1: NULL, arg2: NULL, result_ref: "LABEL_MY_FUNC_END"
```

Nach der Funktionsdefinition werden die Variablen des Hauptteils deklariert und der Funktionsaufruf generiert, der die Werte der Variablen `x` und `y` in die Funktionsargumente `xa` und `yb` setzt. Dabei muss hier schon auf die Argumenttypen, die in der Symboltabelle gespeichert sind, geachtet werden. `:args_1` muss die Adresse für `y` bereits referenzieren, damit spätere Zuweisungen in der Funktion auch die Variable `y` ändern.

Listing 9.18: Drei-Adressen Code Funktionsaufruf

```
ASSIGN, arg1: {int_val: 5}, arg2: NULL, result_ref: "x"
ASSIGN, arg1: {int_val: 3}, arg2: NULL, result_ref: "y"
ASSIGN, arg1: {int_val: 0}, arg2: NULL, result_ref: "z"

ASSIGN, arg1: {var_ref: "x"}, arg2: NULL, result_ref: ":args_0"
REF, arg1: {var_ref: "y"}, arg2: NULL, result_ref: ":args_1"
CALL, arg1: {var_ref: "LABEL_MY_FUNC"}, arg2: NULL, result_ref: "z"
```

Mit diesem Vorlagen können wir nun für alle Konzepte des Slang Quelltextes den Drei-Adressen Code generieren, der optimiert oder direkt weiter an das Compiler-Backend gegeben werden kann.

10 Umsetzung des RISC-V Backends

Damit das Backend gültigen RISC-V Maschinencode generieren kann, muss es dem Drei-Adressen Code und der Symboltabelle der Zwischensprache folgende Dinge entnehmen:

- Register und Speicheradressen, zugeschnitten auf die RISC-V Prozessorarchitektur
- Die RISC Befehle, die die Operationen der Drei-Adressen Operationen abbilden
- Optimierungen, zum Beispiel die Lebensdauer der temporären Variablen, um Register freizugeben

RISC-V hat 31 Allzweckregister, mit denen gearbeitet werden kann. Solange die Anzahl der Variablen und temporären Variablen in einem Programm nicht die Anzahl der Register übersteigt, können diese einfach auf die Register verteilt werden.
Werden mehr Variablen benötigt, kann man zum einen Optimierungen einbauen, die die Lebensdauer der Variablen analysieren und Register für andere Variablen freigeben; zum anderen kann man mit Methoden in der Registerzuteilung arbeiten, um die Variablen in Adressen im Stack zu speichern und, wenn wieder benötigt, in ein Register zu laden.
Da dies das Compiler-Backend deutlich verkompliziert, werden hier nur die Register benutzt, um die Werte der Variablen der Drei-Adressen Operationen zu speichern.

10.1 Register und Speicheradressen

Im ersten Schritt wird also den Variablen in der Symboltabelle Register zugewiesen.
Wenn mehr als 31 temporäre Variablen und Variablen gesetzt sind, brechen wir die Kompilation ab und geben eine Fehlermeldung aus.
Wir verwenden `x0` nicht, da es immer den Wert 0 hat und somit nicht als Register für Variablen genutzt werden kann.

Variablenname	Register
x	x1
foobar	x2
:t1	x3
:t2	x4
y	x5
:t3	x6

Tab. 4: Zuweisung von Variablen in einer fiktiven Symboltabelle zu Registern

10.2 Zuweisung von Drei-Adressen Code zu RISC-V Befehlen

10.2.1 Arithmetische Operationen

Da die Variablen bereits in Register abgelegt sind, können die arithmetischen Operationen der Drei-Adressen Operationen direkt in RISC-V Befehle umgewandelt werden.
Der Befehl `li` (Load Immediate) wird benutzt, um Konstanten in Register zu laden und somit die Register unserer Variablen zu initialisieren.
Mit den Befehlen `add`, `sub`, `mul`, `div` und `rem` werden die arithmetischen Operationen durchgeführt.
Um den Wert einer Variablen mit einer anderen zu überschreiben, wird der Befehl `add` benutzt, um `<reg> + 0` in das zu überschreibende Register zu speichern.

Beispiel Drei-Adressen Code

```
ASSIGN, arg1: {int_val: 5}, arg2: NULL, result_ref: "x"
ASSIGN, arg1: {int_val: 0}, arg2: NULL, result_ref: "y"

ADD arg1: { var_ref: "x" }, arg2: { int_val: 2 }, result_ref: "x"
SUB arg1: { var_ref: "x" }, arg2: { int_val: 1 }, result_ref: "x"
MUL arg1: { var_ref: "x" }, arg2: { int_val: 2 }, result_ref: "x"
DIV arg1: { var_ref: "x" }, arg2: { int_val: 3 }, result_ref: "x"
MOD arg1: { var_ref: "x" }, arg2: { int_val: 4 }, result_ref: "x"

ASSIGN, arg1: {int_val: x}, arg2: NULL, result_ref: "y"
```

Mit folgenden Registern:

Variablenname	Register
x	x1
y	x2

Resultierende RISC-V Befehle

```
li x1, 5
li x2, 0

add x1, x1, 2
sub x1, x1, 1
```

```
mul x1, x1, 2
div x1, x1, 3
rem x1, x1, 4

add x2, x1, 0
```

10.2.2 Sprünge

Bedingte Sprünge können in RISC-V direkt mit dem Befehl `beq` (Branch if Equal) und `bne` (Branch if Not Equal) umgesetzt werden.
Unbedingte Sprünge können mit dem Befehl `jal` (Jump and Link) umgesetzt werden, wobei von `jal` zurückgegebene *Return Address* in `x0` gespeichert und somit ignoriert wird.

Beispiel Drei-Adressen Code

```
ASSIGN, arg1: {int_val: 5}, arg2: NULL, result_ref: "x"
ASSIGN, arg1: {int_val: 10}, arg2: NULL, result_ref: "y"

LABEL, label: "LABEL_S"
COND_GOTO arg1: { var_ref: "x" }, arg2: { var_ref: "y" }, target: "LABEL_T"
COND_NOT_GOTO arg1: { var_ref: "x" }, arg2: { int_val: 5 }, target: "LABEL_T"

LABEL, label: "LABEL_T"
GOTO target: "LABEL_S"
```

Mit folgenden Registern:

Variablenname	Register
x	x1
y	x2

Resultierende RISC-V Befehle

```
li x1, 5
li x2, 10

LABEL_S:
beq x1, x2, LABEL_T
bne x1, 5, LABEL_T
```

```
LABEL_T:
jal x0, LABEL_S:
```

10.2.3 Vergleichsoperationen

Vergleichsoperationen sind durch die geringe Anzahl der Befehlsoperationen etwas interessanter. Der Befehl `slt` (Set Less Than) kann für den `<` (und mit umgedrehten Argumenten für den `>`) Operator benutzt werden. RISC-V implementiert kein Befehl für `<=` und `>=`, dies wird gelöst, indem der `slt` Operator benutzt wird und das Ergebnis mit dem Operator `xori` negiert wird.

$$\textit{Less Or Equal Than} = \texttt{!(x < y)} = \texttt{x >= y}$$
$$\textit{More Or Equal Than} = \texttt{!(y < x)} = \texttt{x <= y}$$

Ansonsten werden im Allgemeinen die Befehle `beq` (Branch if Equal) für `==` und `bne` (Branch if Not Equal) für `!=` empfohlen, um Werte zu vergleichen. Will man dabei einen Wert setzen, kann man dies nach dem entsprechenden Sprungbefehl machen.

Beispiel Drei-Adressen Code

```
ASSIGN, arg1: {int_val: 5}, arg2: NULL, result_ref: "x"
ASSIGN, arg1: {int_val: 10}, arg2: NULL, result_ref: "y"

LT arg1: { var_ref: "x" }, arg2: { var_ref: "y" }, result_ref: "t1"
GT arg1: { var_ref: "x" }, arg2: { var_ref: "y" }, result_ref: "t2"
LE arg1: { var_ref: "x" }, arg2: { var_ref: "y" }, result_ref: "t3"
GE arg1: { var_ref: "x" }, arg2: { var_ref: "y" }, result_ref: "t4"
EQ arg1: { var_ref: "x" }, arg2: { var_ref: "y" }, result_ref: "t5"
NE arg1: { var_ref: "x" }, arg2: { var_ref: "y" }, result_ref: "t6"
```

Mit folgenden Registern:

Variablenname	Register
x	x1
y	x2
t1	x3
t2	x4
t3	x5
t4	x6
t5	x7
t6	x8

Resultierende RISC-V Befehle

```
li x1, 5
li x2, 10

slt x3, x1, x2

slt x4, x2, x1

slt x5, x2, x1
xori x5, x5, -1

slt x6, x1, x2
xori x6, x6, -1

beq x1, x2, LABEL_EQ_NOT_TRUE
  li x7, 0
  j LABEL_EQ_END
LABEL_EQ_NOT_TRUE:
  li x7, 1
LABEL_EQ_END:

bne x1, x2, LABEL_NE_NOT_TRUE
  li x8, 0
  j LABEL_NE_END
LABEL_NE_NOT_TRUE:
  li x8, 1
LABEL_NE_END:
```

RISC-V bietet außerdem einige Pseudo-Operationen an, die zur Lesbarkeit in der Maschinensprache benutzt werden können. Statt `xori x5, x5, -1` kann man die Pseudo-Operation `not`

`x5, x5` benutzen, die zu `xori` expandiert wird, aber etwas lesbarer ist.

10.2.4 Funktionsaufrufe

Funktionsaufrufe können wieder mit dem Befehl `jal` umgesetzt werden, allerdings muss hierbei die *Return Address* in einem Register gespeichert werden, um nach dem Funktionsaufruf wieder zurückzuspringen.
Das macht allerdings verschachtelte Funktionsaufrufe schwierig, da die *Return Address* bei mehreren Funktionsaufrufen ineinander überschrieben wird.
Die Lösung hierfür wäre, die Variablen sauber auf Stack-Adressen zu speichern und die Funktionsdefinition mittels eines Funktionsprologs korrekt aufzusetzen. Dies ist im Rahmen dieser Arbeit zu aufwändig und wird daher nicht implementiert.
Stattdessen wird als temporäre Maßnahme die Kompilierung mit einem Fehler abgebrochen, wenn ein Funktionsaufruf in einem Funktionsaufruf auftritt.
Die *Return Address* wird in `x31` gespeichert, das Register wird dabei als belegt aufgenommen.
Der Wert, der von der Funktion zurückgegeben wird, wird in dem Register `x30` gespeichert.

Beispiel Drei-Adressen Code

Listing 10.1: Drei-Adressen Code Funktionsaufruf

```
ASSIGN, arg1: {int_val: 5}, arg2: NULL, result_ref: "x"
ASSIGN, arg1: {int_val: 3}, arg2: NULL, result_ref: "y"
ASSIGN, arg1: {int_val: 0}, arg2: NULL, result_ref: "z"

ASSIGN, arg1: {var_ref: "x"}, arg2: NULL, result_ref: ":args_0"
REF, arg1: {var_ref: "y"}, arg2: NULL, result_ref: ":args_1"
CALL, arg1: {var_ref: "LABEL_MY_FUNC"}, arg2: NULL, result_ref: "z"
```

Mit folgenden Registern:

Variablenname	Register
x	x1
y	x2
z	x3
:args_0	x4
:args_1	x5
Return Value	x30
Return Address	x31

Resultierende RISC-V Befehle

```
li x1, 5
li x2, 3
li x3, 0

add x4, x1, 0
add x5, x2, 0
jal x31, LABEL_MY_FUNC
add x3, x30, 0
```

10.2.5 Funktionsdefinitionen

Um die Rücksprünge von Funktionsaufrufen zu ermöglichen, ohne Funktionsprolog und -epilog zu implementieren, wird das Register `x31` für die Rücksprungadresse benutzt, wobei verhinder wird, dass der Quellcode Funktionsaufrufe verschachtelt.
Um zu der Adresse zurückzuspringen, die vor dem Funktionsaufruf gespeichert wurde, wird der Befehl `jr` (Jump Return) mit dem Register `x31` benutzt[2].
Diese simple RISC Funktionsdefinition ist strukturell ähnlich zu dem Drei-Adressen Code, aus dem sie generiert wurde.

Beispiel Drei-Adressen Code

Listing 10.2: Drei-Adressen Code Funktionsdeklaration

```
GOTO, arg1: NULL, arg2: NULL, result_ref: "LABEL_MY_FUNC_END"

LABEL, arg1: NULL, arg2: NULL, result_ref: "LABEL_MY_FUNC"
ASSIGN, arg1: {var_ref: ":args_0"}, arg2: NULL, result_ref: ":t1"
REF, arg1: {var_ref: ":args_1"}, arg2: NULL, result_ref: ":t2"

ASSIGN, arg1: {int_val: 20}, arg2: NULL, result_ref: ":t1"
ASSIGN, arg1: {int_val: 10}, arg2: NULL, result_ref: ":t2"

// if-Bedingung
EQ, arg1: {var_ref: ":t1"}, arg2: {int_val: 20}, result_ref: ":t3"
COND_NOT_GOTO, arg1: {var_ref: ":t3"}, arg2: {int_val: 0}, result_ref: "L1"
RET, arg1: {var_ref: ":t1"}, arg2: NULL, result_ref: NULL
LABEL, arg1: NULL, arg2: NULL, result_ref: "L1"

// Standard Return falls nichts zurückgegeben wird
RET, arg1: {int_val 0}, arg2: NULL, result_ref: NULL

LABEL, arg1: NULL, arg2: NULL, result_ref: "LABEL_MY_FUNC_END"
```

Variablenname	Register
:args_0	x4
:args_1	x5
:t1	x6
:t2	x7
:t3	x8
Return Address	x31

Resultierende RISC-V Befehle

```
jal x0, LABEL_MY_FUNC_END

LABEL_MY_FUNC:
add x6, x4, 0

li x6, 20
li x5, 10

beq x6, 20, L1

add x30, x6, 0
jr x31

L1:
li x30, 0
jr x31

LABEL_MY_FUNC_END:
```

Teil III

Auswertungen und Fazit

53

11 Ergebnisse

11.1 Aspekt der Generalität

Der Compiler `SLICC` trennt als Zweiphasen-Compiler den Programmcode von Compilerfrontends und Compilerbackends, was es leicht macht neue Sprachen zu parsen und Maschinencode für neue Prozessorarchitekturen zu generieren.
Allerdings heißt es nicht, dass Frontend von Backend komplett unabhängig sind; Die Zwischensprache spielt eine größere Rolle als erwartet. Ist sie zu einfach, müssen Frontends viele TACs generieren und werden durch Limitationen der Symboltabelle in der Generalität des Quelltextes beschränkt. Man verliert potenziell an Semantik in der Zwischensprache und es ist schwieriger, generische Optimierungen zu sichten und einzubauen.
Dies gilt auch für die Backends, die, um die Operationen der Zielprozessorarchitektur ausnutzen zu können, mittels Optimierung die einfachen Operationen in den TACs gruppieren müssen anstatt für jede Operation eine Vorlage an Maschinencode definieren zu können.

Ist die Zwischensprache zu komplex, wird die Rolle der Optimierung verschoben. Die Frontends müssen dann, beim parsen des Quelltextes, syntaktisch ähnliche Konstrukte unterscheiden um die korrekten TACs zu generieren statt nur einen Teil der Operationen der Zwischensprache zu nutzen.

Scheinbar ist es von Vorteil für die Implementationskomplexität der Frontends eine simple Zwischensprache zu haben. Für Optimierungen der Ausgabe scheint es aber wiederum besser, die Zwischensprache komplexer zu gestalten. Hier muss man einen Kompromiss finden, der die Implementierung der Frontends nicht zu komplex macht, aber auch genug Informationen für die Optimierung der Ausgabe bereitstellt.
Eine Alternative zum Kompromiss wäre, die Zwischensprache in mehrere Schichten aufzuteilen, die aufeinander aufbauen.
Ein Frontend kann so in simple Konstrukte der Zwischensprache 1 parsen, die vom Hauptteil des Compilers in eine komplexere und optimierte Zwischensprache 2 umgewandelt wird. Die Backends können dann auf dieser Zwischensprache 2 arbeiten und die Operationen direkt in komplexeren Maschinencode umwandeln.
Damit vermeidet man zumindest, Optimierungen für viele Backends zu duplizieren, erhöht aber wiederrum die Komplexität der Schnittstelle.

Der TAC für den `SLICC` Compiler ist simpel gehalten, was zu der RISC Zielarchitektur passt. Neue Frontends und Backends können durch die kleine Schnittstelle leicht hinzugefügt werden, Optimierungen für die Zielarchitektur sind aber schwerer zu implementieren.

Diese Abwägung passt meiner Meinung nach zum Ziel des Compilers, eine Grundlage für diese Arbeit zu sein und als Lernbasis für den Compilerbau darzustellen.

11.2 Das Slang Frontend

Mithilfe der Bibliotheken `Flex` und `Bison` gelang es, ein einfaches und erweiterbares Frontend für die Programmiersprache `Slang` zu implementieren.
Verständliche Fehler im Bottom-Up Parser anzuzeigen ist trivial. Regelübergreifende Semantik aus dem Syntaxparser abzuleiten ist dagegen mit mehr Aufwand verbunden. Da das Frontend aber zu einer simplen Zwischensprache parst, können viele der theoretisch notwendigen Semantiken (wie zum Beispiel die Lebenszeit von Variablen) auf spätere Optimierungsschritte an der Zwischensprache ausgelagert werden.

11.3 Die Zwischensprache

Die Implementation der Symboltabelle benötigt noch eine genauere Definition, wie verschachtelte Funktionsaufrufe dargestellt werden können.
Unter Umständen ist eine einzelne `parent` Referenz genug, um die Verschachtelung zu repräsentieren. Allerdings muss in der Praxis genauer untersucht werden, ob die Zielarchitekturen damit umgehen können oder mehr Informationen benötigen.

Die Definition des Drei-Adressen Codes ist momentan passend auf die Zielarchitektur RISC zugeschnitten; der Teil der Funktionsparameterübergabe mit hart definierten Symbolen und der Übergabe dieser an die aufgerufenen Funktionen scheint mir aber umständlich und könnte vereinfacht werden.
Hier wäre es sinnvoll genauer auf die Anforderungen der Zielarchitektur einzugehen und die entsprechenden Operationen anzupassen, bevor weitere Backends gebaut werden.

11.4 Das RISC-V Backend

Die Entscheidung, auf Variablen im Stack zu verzichten, nur Register zu benutzen und somit das Backend zu “vereinfachen”, hat zu mehr Problemen als erwartet geführt. Nicht nur, das Programme dadurch in der Anzahl der Variablen begrenzt sind, es kann auch nicht mehr als eine Ebene des Funktionsaufrufs behandeln.
Dadurch wird das Backend relativ nutzlos und kann nur triviale Programme umsetzen. Hier

würde ich definitiv die Variablen im Stack halten und die Register nur für die Optimierung der Operationen benutzen.

11.5 Fazit

Einen simplen, unoptimierten Zweiphasen-Compiler für eine simple Programmiersprache von Grund auf zu implementieren ist algorithmisch weniger anspruchsvoll als ich erwartet habe. Der Weg von hoher Programmiersprache zu gültigem Maschinencode und die Umformung der Operationen ist nicht, wie anfangs erwartet, eine "BlackBox" die schwierig zu durchschauen ist.
Stattdessen sind die scheinbar kleinen Entscheidungen, wie Datenstruktur der Zwischensprache, ein entscheidender Teil der Komplexität und Interoperabilität der Compilerkomponenten. Da hier für generische Compiler in vielen Entscheidungen Kompromisse gefunden werden müssen, ist Praxiserfahrung ein entscheidender für die Lesbarkeit, Nutzbarkeit und Erweiterbarkeit des Compilers.

Mir hat die Arbeit an dem Compiler das Interesse an der Arbeit und Optimierung von Maschinencode geweckt. Ich werde mich in Zukunft weiter mit dem Compiler `SLICC` beschäftigen und die aufgelisteten Probleme zu beheben sowie Optimierungen an der Generierung des Maschinencodes vorzunehmen.

Literaturverzeichnis

[1] Aho, Alfred V. ; Lam, Monica S. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2nd Edition).* USA : Addison-Wesley Longman Publishing Co., Inc., 2006. – ISBN 0-321-48681-1

[2] Andrew Waterman ; Krste Asanovic: *The RISC-V Instruction Set Manual Volume I: User-Level ISA.* Mai 2017

[3] Calingaert, Peter: *Assemblers, Compilers, and Program Translation.* USA : W. H. Freeman & Co., 1979. – ISBN 0-914894-23-4

[4] Free Software Foundation, Inc.: *GNU Bison Manual*

[5] Meduna, A.: *Elements of Compiler Design.* CRC Press, 2007. – ISBN 978-1-4200-6325-7

[6] Mössenböck, H.: *Compilerbau: Grundlagen Und Anwendungen.* dpunkt.verlag, 2024 (Lehrbuch). – ISBN 978-3-9889014-6-0

[7] Pascal Ernst: *SLICC.* https://github.com/LinuCC/simple-language-to-instruction-compiler/tree/main/slicc. Oktober 2024

[8] Paul Kline: *BNF Playground.* https://bnfplayground.pauliankline.com/

[9] Paxson, Vern: *Flex 2.5.4 Manual.* März 1995

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere, dass ich alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe, und dass die eingereichte Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist.

Suderburg, den 9. Oktober 2024

Ort, Datum

Unterschrift

58

Anhang

59

Anhang A

Beispielcode in Slang

```
program my_calculations {
  int i = 0;
  int y;
  int foo = 5;
  int[5] bar;
  int barsize = 5;

  foo = i + 5;

  for (y < barsize; y = y + 1) {
    bar[y] = x + y;
  }
  call_by_value(foo);
  call_by_reference(y);
  if (y == 5) {
   foo = call_with_return(y);
  }
  write(foo);
}

func void call_by_value(int val) {
   write(val);
}

func void call_by_reference(ref int val) {
   read(val);
}

func int call_with_return(int val) {
int x = 2;
return x / 2 + val * 2;
}
```

Anhang B

Komplette EBNF Grammatikdefinition für Slang

```
/* Start der Grammatik, unser Programmcode beginnt mit program { }, */
/* dahinter stehen die Funktionsdefinitionen */
<main> ::= <program> <wso> <func_def>* <wso>

/* Ein Programm besteht aus einem Block */
<program> ::= "program" <wso> <id> <wso> <block>

<block> ::= <wso> "{" <wso> <block_body> <wso> "}" <wso>

/* Ein Block besteht aus optionalen Variablendefinitionen und einer */
/* Anweisungsliste */
<block_body> ::= <variable_list> <wso> <statement_list>
              | <statement_list>

<variable_list> ::= <variable_declaration>
                 | <variable_declaration> <wso> <variable_list>

/* Eine Variable kann deklariert oder mit einem Wert initialisiert werden */
<variable_declaration> ::= ((<type> <wso> <id> <wso> "=" <wso> <intliteral> <wso>) |
    (<type> <wso> <id> <wso>)) ";"

<statement_list> ::= <statement>
                  | <statement> <wso> <statement_list>

/* Ein Statement ist eine Variablenzuweisung, eine Kontrollstruktur, */
/* eine for-Schleife, ein Funktionsaufruf oder ein ‘return‘ */
<statement> ::= <statement_assignment> <wso> ";"
             | <if_statement>
             | <for_loop>
             | <func_call> <wso> ";"
             | "return" <wso> <expression> <wso> ";"

<statement_assignment> ::= <assigment_left> <wso> "=" <wso> <expression> <wso>

/* Eine Variablenzuweisung kann auch einen Array-Eintrag referenzieren */
/* z.B. ‘foo[x] = 1’ */
<assigment_left> ::= (<id>) | (<id> "[" <wso> (<intliteral> | <id>) <wso> "]")
```

```
/* Ein Ausdruck ist eine mathematische Operation, ein Funktionsaufruf oder ein
    Vergleich */
<expression> ::= <intliteral>
              | <id>
              | <func_call>
              | <expression> <wso> "+" <wso> <expression>
              | <expression> <wso> "-" <wso> <expression>
              | <expression> <wso> "*" <wso> <expression>
              | <expression> <wso> "/" <wso> <expression>
              | <compare_expression>
              | "(" <wso> <expression> <wso> ")"

<comparison_operator> ::= "=="
                        | "!="
                        | "<"
                        | "<="
                        | ">"
                        | ">="

/* Erlaube erstmal nur leichte Ausdruecke im Vergleich */
<compare_expression> ::= <id> <wso> <comparison_operator> <wso> <id>
                       | <id> <wso> <comparison_operator> <wso> <intliteral>
                       | <intliteral> <wso> <comparison_operator> <wso> <id>
                       | <intliteral> <wso> <comparison_operator> <wso> <intliteral>

/* Eine if-Kontrollstruktur mit optionalem else */
<if_statement> ::= "if" <wso> "(" <wso> <compare_expression> <wso> ")" <wso> <block>
                 | "if" <wso> "(" <wso> <compare_expression> <wso> ")" <wso> <block>
                     <wso> "else" <wso> <block>

/* Eine for-Schleife ohne Variablendeklaration; diese ist im ersten */
/* Teil des blocks zu definieren */
<for_loop> ::= "for" <wso> "(" <wso> <compare_expression> ";" <wso>
    <statement_assignment> <wso> ")" <wso> <block>

/* Ein Funktionsaufruf mit Argumenten */
<func_call> ::= <id> <wso> "(" <wso> <func_args> <wso> ")" <wso>
<func_args> ::= <expression>
              | <expression> <wso> "," <wso> <func_args>

/* Eine Funktionsdefinition */
<func_def> ::= "func" <wso> (<type> | "void") <wso> <id> <wso> "(" <wso>
    <func_args_def> <wso> ")" <wso> <block>
```

```
<func_args_def> ::= <func_arg_def>
                  | <func_arg_def> <wso> "," <wso> <func_args_def>
<func_arg_def> ::= <type> <wso> <id>
                  | "ref" <wso> <type> <wso> <id> <wso>

/* Typen und Variablen */
<type> ::= "int" | <intarray>
<intarray> ::= "int[" <intliteral> "]"

/* Andere Zeichen */
<digit> ::= [0-9]
<intliteral> ::= <digit>+
<alphachar> ::= [a-z]
<id> ::= <alphachar> (<alphachar> | <digit> | "_")*
<ws> ::= " " | "\t" | "\n"

/* Optionale Leerzeichen oder newlines */
<wso> ::= <ws>*
```

Validiert mit Anhang A auf BNF Playground [8]

Anhang C

SLICC Compiler Handbuch

C.1 Installation

Der aktuelle Programmcode des Compilers ist auf GitHub verfügbar [7].

Um den Compiler zu bauen, wird CMake benötigt. CMake ist ein weit verbreitetes Build-System, das auf vielen Plattformen verfügbar ist. Zusätzlich dazu wird der Kompiler `clang` benötigt sowie die Source Files zu den Bibliotheken `GNU Bison` und `Flex`.

C.1.1 Linux

Die Abhängigkeiten können auf den meisten Linuxdistributionen mit dem Paketmanager installiert werden. Für Debian-basierte Distributionen wie Ubuntu kann dies mit folgendem Befehl getan werden:

```
sudo apt install cmake clang bison flex
```

C.1.2 MacOS

Auf MacOS reicht es, XCode über den App Store zu installieren. XCode kommt mit allen benötigten Tools, um den Compiler zu bauen.

C.2 Bauen des Compilers

Zum Bauen des Binärkompilates führt folgender Befehl:

```
cmake .; make;
```

Mit dem Flag `DCMAKE_BUILD_TYPE=Debug` wird der Compiler im Debug-Modus gebaut:

```
cmake -DCMAKE\_BUILD\_TYPE=Debug .; make;
```

Bei erfolgreicher Kompilierung wird ein ausführbares Programm namens `slicc` im Projektverzeichnis erstellt.

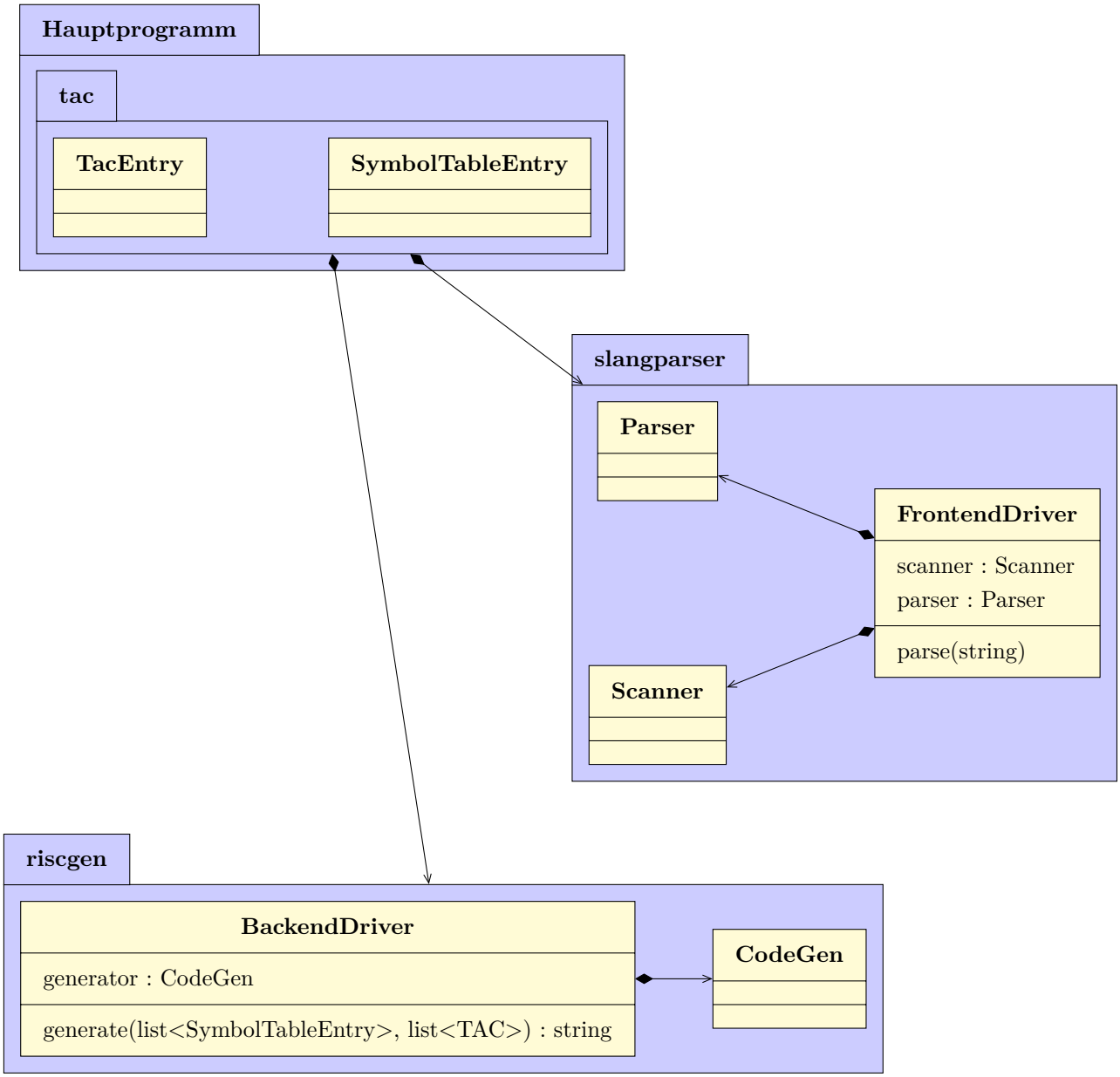
C.3 Benutzung des Compilers

Der Compiler liest den zu kompilierenden Quelltext aus Standard Input und schreibt den generierten Maschinencode in den Standard Output.

Um die Datei `examples/simple_test.slang` mit SLICC zu kompilieren kann man diese in den Compiler pipen:

```
cat examples/simple\_test.slang | ./slicc
```

C.4 Strukturübersicht



Der Compiler besteht aus drei Komponentenarten: dem Frontend, dem Backend und dem Hauptprogramm. Es können mehrere Frontends und Backends existieren, die von dem Hauptprogramm importiert und benutzt werden können.
Der `*Driver` in der jeweiligen Komponente ist die Klasse, die von `main()` benutzt wird, um die Funktionalität der Komponente zu nutzen.

Beim Kompilieren wird der Quelltext von `main()` eingelesen und an den `FrontendDriver` in der `slang_parser` Komponente übergeben. Diese parst den Quelltext mittels der von Flex und GNU Bison autogenerierten Klassen `Scanner` und `Parser`. Die Zwischensprache, bestehen aus den Symboltabelleneinträgen und TACs, wird dann von `main()` an den `BackendDriver` in der `risc_gen` Komponente übergeben. Die Klasse `CodeGen` generiert dann den Maschinencode als String aus den Symboltabelleneinträgen und TACs und gibt sie an `main()` zurück, welche sie dann ausgibt.