Maturaarbeit

Entwicklung und Auswertung eines alternativen Aufbaus für Compiler

Fabio Stalder

Betreut durch
Thomas Jampen

10. Oktober 2024

Gymnasium Kirchenfeld
Abteilung MN

Inhaltsverzeichnis

Während ich mich mit der Theorie hinter modernen Compilern befasste, stellte ich mir häufig die Frage: Wieso sind Compiler so, wie sie sind? Ich nahm mir daraufhin vor, selbst einen alternativen Aufbau für Compiler zu entwickeln, mit dem Ziel zu verstehen, wieso sich das traditionelle Compilermodell so gut bewährt. In dieser Maturaarbeit werde ich zuerst den traditionellen Aufbau von Compilern kurz beschreiben und mein eigenes alternatives Modell vorstellen. Die Umsetzung meiner Idee werde ich anhand eines selbstgeschriebenen alternativen Compilers erläutern. Dieser alternative Compiler wird daraufhin mit zwei traditionellen Compilern, wovon ich ebenfalls einen selbst geschrieben habe, verglichen. Zum Schluss werde ich die Resultate des Vergleichs einordnen und daraus Vor- und Nachteile des traditionellen Compileraufbaus ableiten.

1 Einleitung

In der Informatik beschreibt Compiler ein Programm, das Code aus einer Programmiersprache in eine tiefere Programmiersprache übersetzt. In dieser Hinsicht gleichen Compiler Übersetzern für Menschensprache. Genau wie ein Übersetzer für die Kommunikation zwischen zwei verschiedensprachigen Menschen nötig ist, braucht man Compiler, um die Kommunikation zwischen Mensch und Computer zu ermöglichen oder zumindest zu vereinfachen. Grundsätzlich ist es mit einer Assembly-Sprache möglich, ohne Compiler einem Computer Befehle zu geben. Jedoch sind Assembly-Sprachen nicht besonders einfach zu verwenden. Compiler übersetzten von für Menschen verständlicheren Programmiersprachen zu Computersprachen und ermöglichen daher ein viel einfacheres Schreiben von Programmen. Compiler unterscheiden sich jedoch grundlegend von Übersetzern in der Erwartungshaltung, die an sie gestellt wird. Menschensprache ist sehr komplex und nicht immer eindeutig. Programmiersprachen hingegen sind so definiert, dass sie keinen Raum für Missverständnisse oder Ungenauigkeit lassen. Genauso muss auch ein Compiler exakt und fehlerfrei übersetzen. Neben fehlerfrei muss die Kompilierung auch möglichst schnell sein. Dasselbe gilt natürlich auch für die resultierende Ausgabedatei. Diese sollte optimal generiert werden, um die schlussendliche Ausführungsdauer so kurz wie möglich zu halten. Sollte sich doch einmal ein Fehler in der Eingabedatei befindet, muss der Compiler diesen verständlich melden. Compiler sind also keine simplen Programme und daher auch bis heute ein aktives Forschungsgebiet.

Als ich mit meiner Maturaarbeit begann, war mein Ziel die Entwicklung eines einfachen Compilers zu einer C ähnlichen Programmiersprache. Unterstützt wurde ich dabei durch eine Vorlesung der Universität Bern [2]. Während mir in dieser Vorlesung der theoretische Aufbau eines Compilers erklärt wurde, fragte ich mich trotzdem hin und wieder, wieso genau Compiler so aufgebaut sind und ob es nicht eine einfachere Möglichkeit gäbe. Als ich dann in den Sommerferien auf Probleme in der Entwicklung meines eigenen Compilers stiess, entschied ich mich, einem von mir erdachten alternativen Aufbau für Compiler eine Chance zu geben.

In dieser Maturaarbeit werde ich anhand des "traditionellen" Compileraufbaus meine alternative Idee erklären und auf mögliche Probleme in der Umsetzung eingehen. Zum Schluss werde ich einen nach dem alternativen Aufbau entwickelten Compiler mit traditionellen Compilern vergleichen und damit Vor- und Nachteile der beiden Möglichkeiten aufzeigen. Für diese Maturaarbeit werden Grundkenntnisse von Regular-Expressions, Datenstrukturen und Assembly vorausgesetzt.

2 Vergleich der Compiler

Um meinen alternativen Compileraufbau zu testen, werde ich folgende drei Compiler vergleichen:

1. Der QHScompiler ist der von mir nach meinem Aufbau entwickelte Compiler. Seine genaue Funktionsweise wird in Kapitel 4 ausgeführt. Er ist in C++ geschrieben und verwendet meine eigene Programmiersprache QHS als Eingabesprache.

2. Bei der Veröffentlichung im Jahr 1987 stand das Akronym GCC noch für GNU C Compiler. Heute ist GCC die GNU Compiler Collection, eine Sammlung an Compilern für verschiedene Programmiersprachen, darunter C, C++, Rust, etc. Für diesen Vergleich verwende ich den C Compiler aus der GNU Compiler Collection (Version 12.4). Jede weitere Verwendung der Abkürzung GCC referiert auf diese Version des C Compilers. Für diesen Vergleich repräsentiert GCC den traditionellen Compileraufbau.

3. Der THScompiler ist ebenfalls ein traditioneller Compiler. Der Unterschied zu GCC liegt jedoch darin, dass der THScompiler von mir selbst entwickelt wurde. Er ist dadurch deutlich weniger optimiert und umfasst weniger Funktionen. In der Komplexität entspricht der THScompiler ungefähr dem QHScompiler. Der THScompiler dient mit ähnlichem Arbeitsaufwand, Optimierung und Niveau der Programmierung als "realistische" Konkurrenz zum QHScompiler. Geschrieben ist der THScompiler in C++ und kompiliert aus meiner eigenen Programmiersprache THS.

An die von mir entwickelten Compiler habe ich folgende Mindestanforderungen gestellt:

Tabelle 2.1: Anforderungen an die Compiler

Ausgabe als Assembly-Code	Die Ausgabesprache muss x86 Assembly sein
C ähnlicher Syntax	Die Eingabesprache muss einen C ähnlichen Syntax aufweisen
Variablen und Funktionen	Lokale und globale Variablen sowie Funktionen müssen unterstützt werden
Verzweigungen und Schleifen	Verzweigungen und Schleifen müssen umsetzbar sein

Dabei sollen die Compiler in folgenden Kriterien verglichen werden:

Tabelle 2.2: Vergleichskriterien der Compiler

Geschwindigkeit der Kompilierung	Wie lange dauert die Kompilierung?
Geschwindigkeit der Ausgabedatei	Wie lange dauert die Ausführung der Ausgabedatei?
Umgang mit fehlerhaftem Code	Wie geht der Compiler mit Fehlern in der Eingabedatei um?
Offenheit für Erweiterung	Wie einfach kann die Eingabesprache erweitert werden?

3 Funktionsweise traditioneller Compiler

Der traditionelle Aufbau eines Compilers lässt sich mit folgendem Schema veranschaulichen:

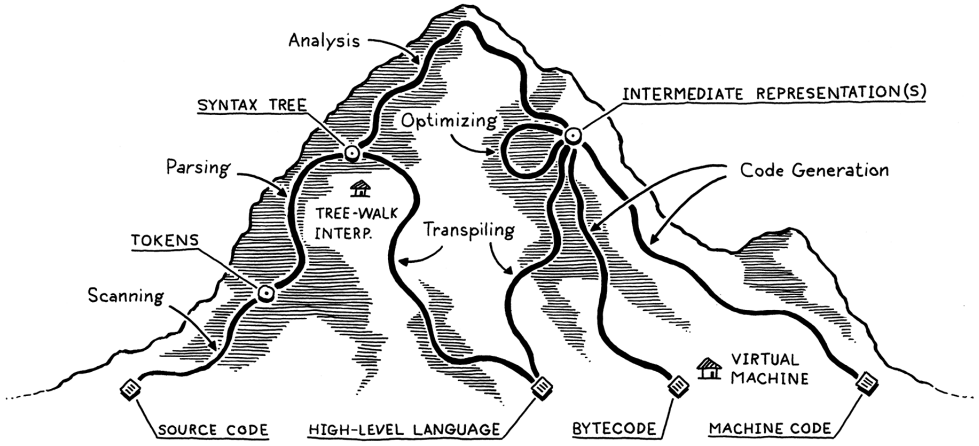

Abbildung 3.1: Schritte, die ein Compiler durchläuft [4]

Ein traditioneller Compiler beginnt unten links bei der Eingabedatei (*Source Code*). Über verschiedene Wege wird die Ausgabedatei als *High-level Language*, *Bytecode* oder *Machinecode* generiert. In dieser Arbeit werde ich mich lediglich auf die in der unteren Abbildung 3.2 dargestellten Schritte fokussieren.

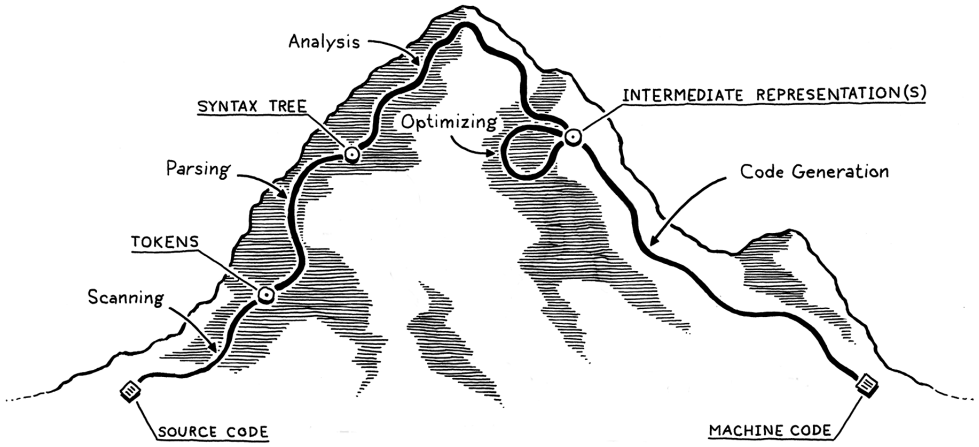

Abbildung 3.2: Schritte, die in dieser Arbeit behandelt werden (Basierend auf [4])

Die in dieser Maturaarbeit verwendeten Fachbegriffe entsprechen nicht immer denen aus Abbildung 3.2. Bei der ersten Verwendung dieser Fachbegriffe wird jeweils die alternative aus Abbildung 3.2 in Klammern angegeben.

Als Basis für diese Kapitel dient das Buch *Compilers: Principles, Techniques, and Tools (2nd Edition)* [1]

und eine Vorlesung der Universität Bern zum Thema Compiler Construction [2].

3.1 Lexikalische Analyse (*Scanning*)

Meist werden Programme so geschrieben, dass wir Menschen sie lesen und verstehen können. Dafür verwendet man Buchstaben, Zahlen, Leerzeichen und Sonderzeichen (wie + oder ∗). Diese Zeichen sind jedoch für den Computer nicht sofort verständlich. Der erste Schritt beim Kompilieren ist daher die lexikalische Analyse. Diese Analyse wird von einem Teil des Compilers, dem *Lexer*, durchgeführt. Die Aufgabe des *Lexers* ist es, die Eingabedatei abzuarbeiten und die gefundenen Zeichen in sogenannte *Tokens* zu verwandeln. Diese *Tokens* sind Datenstrukturen, die der Compiler versteht und mit denen er weiterarbeiten kann.

Ein Beispiel der lexikalischen Analyse für die der Programmiersprache C:

Auflistung 3.1: C code vor lexikalischer Analyse

```
int foo()
{
    if (bar == 0)
    {
        return 0;
    }

    return 1;
}
```

Auflistung 3.2: *Tokens* nach lexikalischer Analyse

```
Keyword         (keyword="int")
Identifier      (id="foo")
LParenthesis
RParenthesis
Keyword         (keyword="if")
LParenthesis
Identifier      (id="bar")
Operator        (operator=ComparisonEqual)
LiteralInt      (value=0)
[...]
```

Der *Lexer* legt fest, welche Zeichen die Eingabesprache enthalten darf und welche Bedeutung ihnen zugesprochen wird. So ist zum Beispiel im *Lexer* festgelegt, dass ein + Zeichen als Addition interpretiert wird. Genauso wie im Listing 3.2 `'if'` als *Keyword-Token* gesehen wird, lässt sich im *Lexer* auch bestimmen, dass ein Wort wie `'else'` als Keyword angesehen werden soll.

3.2 Syntaktische Analyse (*Parsing*)

Der Compiler hat nun die Zeichen der Eingabedatei in ein für ihn verständliches Format übersetzt. Jedoch fehlt dem Compiler noch das Verständnis für die Syntax der Eingabe-Programmiersprache. Die meisten High-Level Programmiersprachen weisen Syntaxregeln auf. Diese beinhalten, wie Funktionen und Variablen

definiert werden oder mit welchen Präzedenzregeln mathematische Ausdrücke evaluiert werden. In diesem Schritt führt der sogenannte *Parser* die syntaktische Analyse durch. Dabei werden die bei der lexikalischen Analyse gefundenen *Tokens* ineinander verschachtelt und in einen sogenannten *Abstract Syntax Tree* (AST) überführt.

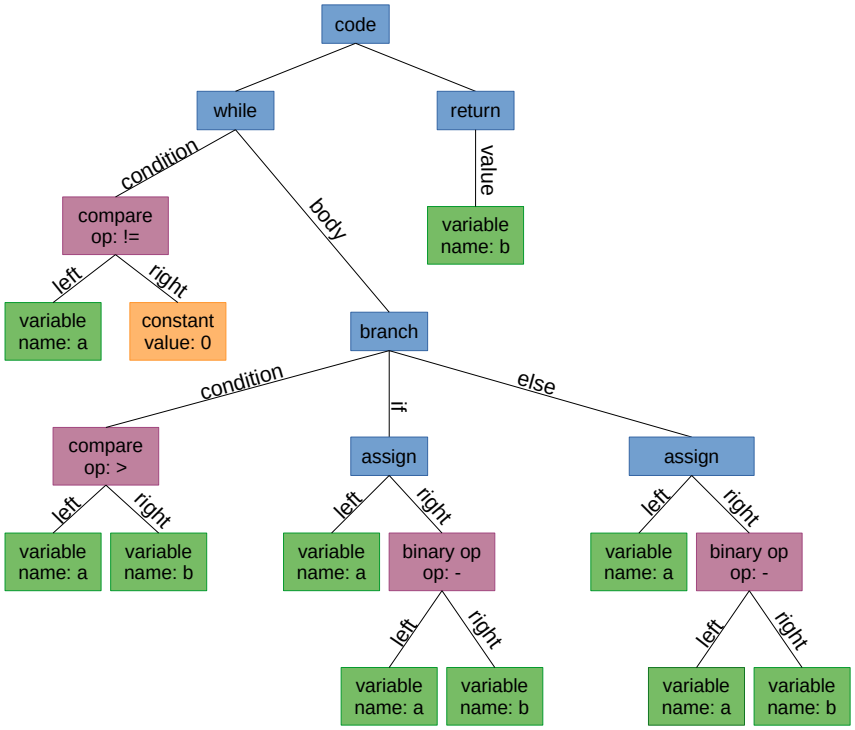

Abbildung 3.3: AST zum euklidischen Algorithmus (Basierend auf [5])

Ein AST enthält somit nicht nur Informationen über die *Tokens*, sondern über die gesamten Strukturen und Abhängigkeiten, die sich aus den *Tokens* ergeben. Variabel- und Funktionsdefinitionen oder komplexe *Statements* und *Expressions* sind im AST als *Nodes* enthalten. Wenn man die *Nodes* des AST von unten nach oben durchquert, erhält man die Reihenfolge der einzelnen *Tokens* ohne Abhängigkeitskonflikte. Eine Subtraktion kann zum Beispiel erst ausgeführt werden, wenn sowohl die linke als auch die rechte Zahl bekannt ist. Daher befindet sich, wie in Abbildung 3.3 ersichtlich, die Subtraktion über den beiden benötigten Werten im AST.

3.3 Semantische Analyse (*Analysis*)

Semantik ist die Wissenschaft der Bedeutung von Wörtern einer Menschensprache. Bei einem Compiler geht es bei der semantischen Analyse jedoch nicht um die Bedeutung, sondern um die Korrektheit von *Expressions*. Wird eine Variable nicht konform ihres Datentyps verwendet, zum Beispiel wenn zwei Strings dividiert werden sollen, wird dies während der semantischen Analyse entdeckt und gemeldet. Gegebenenfalls kann ein impliziter Cast, also eine automatische Umwandlung des Datentyps, hinzugefügt werden.

So geben zum Beispiel manche Programmiersprachen bei der Division zweier Ganzzahlen eine Fliesskommazahl zurück. Auch werden unbekannte Variablen und Funktionen in diesem Schritt abgefangen. Hauptsächlich werden Datentypen an die *Nodes* des AST angebunden. Nach der semantischen Analyse sieht der AST aus Abbildung 3.3 wie folgt aus:

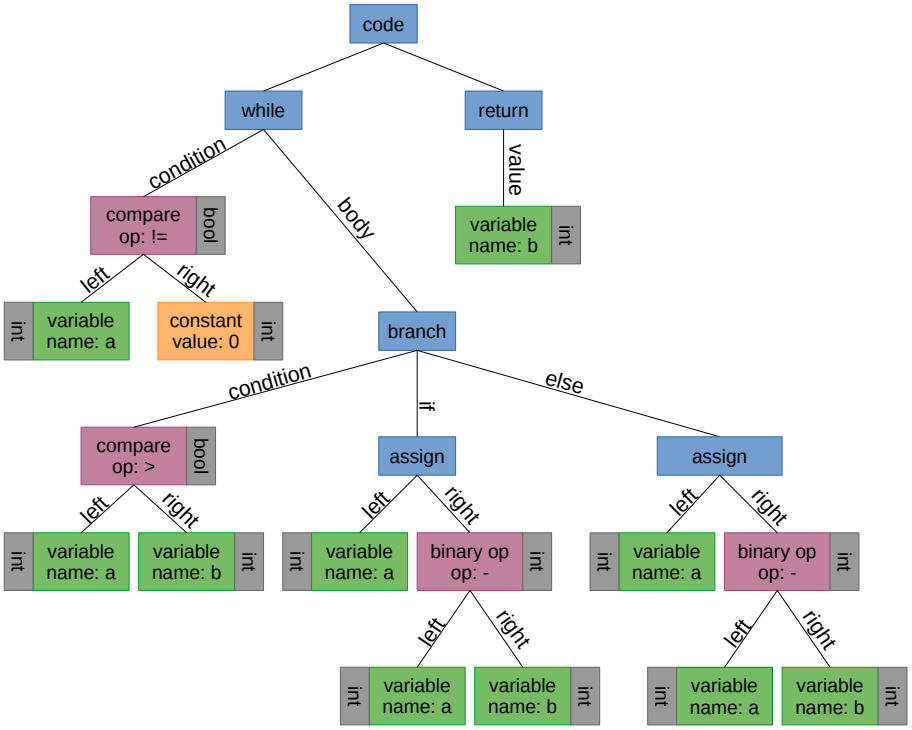

Abbildung 3.4: AST nach der semantischen Analyse (Basierend auf [5])

Der AST wird nach der semantischen Analyse als Zwischencode (*Intermediate Representation*) bezeichnet. Da dieser Zwischencode in den weiteren Beispielen jedoch weiterhin dem Aufbau eines AST folgt, wird auch weiterhin der Begriff AST verwendet.

3.4 Codegenerierung

Codegenerierung ist der finale Schritt, der ein Compiler ausführen muss. Nun da die Eingabedatei als Zwischencode vorliegt, kann die Ausgabedatei generiert werden. Eine geläufige Methode der Codegenerierung ist die sogenannte *Macro Expansion*. Dabei wird der AST von unten nach oben schrittweise mit Teilen an Ausgabecode ersetzt. Diese Ausgabecode-Teile sind häufig von den darunterliegenden *Nodes* abhängig. Der AST aus Abbildung 3.4 sähe nach erfolgreicher *Macro Expansion* wie folgt aus:

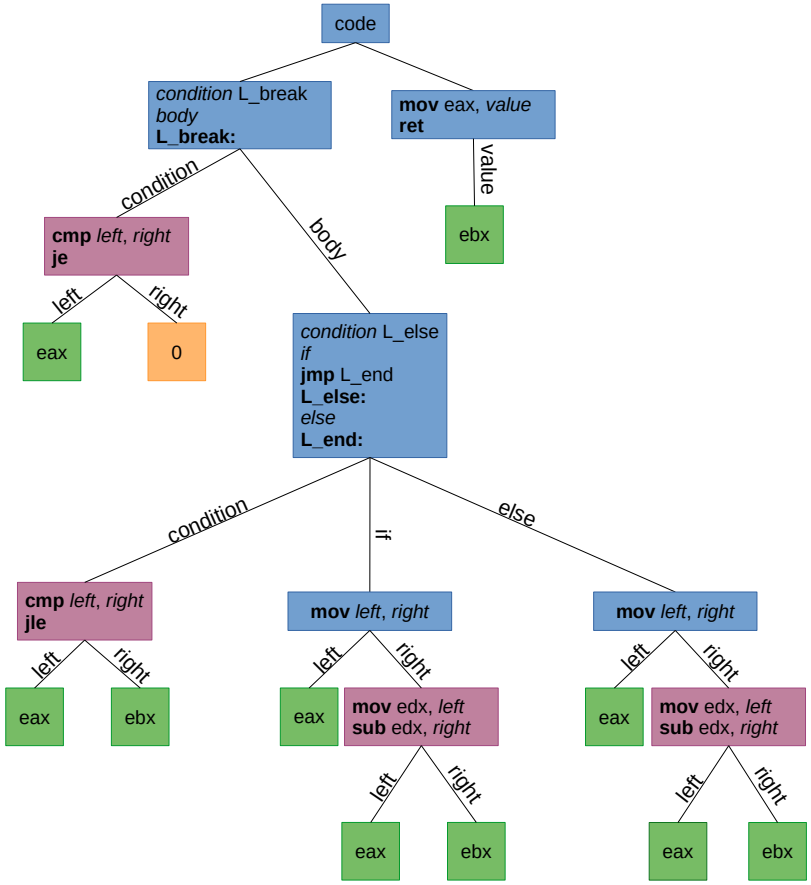

Abbildung 3.5: AST nach *Macro Expansion* (Basierend auf [5])

Nach der *Macro Expansion* lässt die Ausgabedatei einfach aus dem AST generieren.

3.5 Optimierung

Codegenerierung ist zwar der letzte Schritt beim Kompilieren, trotzdem wurde eine wichtige Aufgabe des Compilers noch nicht betrachtet. Optimierung geschieht zwischen jedem der genannten Schritte. Dabei geht es darum die Ausgabe so effizient wie möglich zu machen. Effizient kann hierbei verschiedenes bedeuten. Die Ausgabedatei muss so schnell wie möglich ausgeführt werden, Arbeitsspeicher sparsam verwenden und dazu noch eine möglichst kleine Datei umfassen. Optimierungsmethoden reichen vom Überspringen der Kommentare und Umstellen von mathematischen Operationen, bis zum Entfernen von ungebrauchten Variablen und unnötigem Code. Es muss von CPU-Registern profitiert, mit Heap-Arbeitsspeicher umgegangen und von Inlinefunktionen Gebrauch gemacht werden. Optimierung ist also sehr vielseitig und komplex. Wie Optimierung genau funktioniert, wird in dieser Arbeit nicht weiter betrachtet.

4 Der QHScompiler

Der QHScompiler, wie in Kapitel 2 bereits angesprochen, basiert auf einem von mir erdachten alternativen Aufbau für einen Compiler. Diesem Aufbau liegt eine einfache Idee zugrunde: Die Macros der *Macro Expansion* aus Abschnitt 3.4 sollen erst während der Kompilierung innerhalb der Eingabedatei und nicht im Compiler definiert werden. Auf dieser Grundidee werde ich zwei Dinge aufbauen. Erstens halte ich es für möglich, mit der richtigen Verwendung von Macros die gesamte syntaktische Analyse zu überspringen und keinen AST generieren zu müssen. Zweitens sollte es rein durch die Veränderung von diesen Macros möglich sein jegliche Programmiersprache zu kompilieren. Man könnte also in einem Dokument verschiedene Programmiersprachen verwenden und müsste dazwischen bloss die jeweiligen Macros neu definieren. Das gleiche gilt ebenfalls für die Ausgabesprache. Nur durch das Umdefinieren der Macros liesse sich die Ausgabesprache wechseln, ohne eine Änderung am Compiler vornehmen zu müssen. Um dies zu verwirklichen, unterscheidet sich der QHScompiler stark von traditionellen Compilern.

Die Programmiersprache, in der sich diese Macros definieren lassen, bezeichne ich als QHS. Der Compiler der QHS versteht und kompiliert, wird dazu passend QHScompiler genannt. QHS besteht wie die meisten anderen Programmiersprachen aus Wörtern. Im Kontext von QHS werden diese Wörter *Orders* genannt. Orders können drei verschiedenen Typen aufweisen: *Identifiers*, *Instructions* und *LiteralCode*. Wie diese drei *Ordertypen* genau funktionieren, wird in Abschnitt 4.3 ausführlicher erklärt.

Der Kompilierung durch den QHScompiler liegt ein einfacher Zyklus, der QHS-Zyklus, zugrunde.

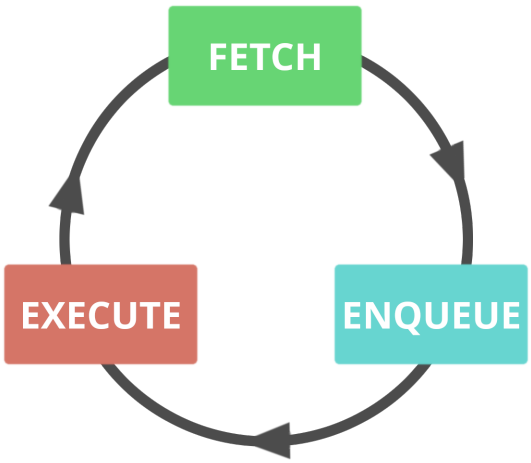

Abbildung 4.1: Zyklus der QHS Kompilierung

4.1 Die *Fetch-Phase*

Die Aufgabe der *Fetch-Phase* ist es die nächste *Order*, die verarbeitet werden soll, zu finden. In dieser Hinsicht gleicht die *Fetch-Phase* der lexikalischen Analyse eines traditionellen Compilers. Der QHS-Zyklus beginnt bei der *Fetch-Phase*, dabei wird die erste *Order* aus der Eingabedatei extrahiert. In jeder weiteren *Fetch-Phase* wird die nächste *Order* aus der Eingabedatei geholt. Die *Fetch-Phase* ist auch dafür zuständig die nächste *Order* einem der drei Typen (*Identifier*, *Instruction* oder *LiteralCode*) zuzuordnen. Diese *Ordertypen* sind mit folgenden Regular-Expressions definiert. Leerzeichen dienen als Trennung zwischen *Orders* und werden ignoriert.

Tabelle 4.1: RegEx Definitionen der *Ordertypen*

identifier	`[^\s]+`
instruction	`#[^\s]+`
literalCode	`".*"`

Im Vergleich zu traditionellen Compilern fällt auf, dass beim QHScompiler kaum zwischen Zeichen differenziert wird. Während die lexikalische Analyse traditionell zwischen vielen verschiedenen *Tokens* unterscheidet, sind für den QHScompiler alle Zeichen (mit Ausnahme von # und ") gleichbedeutend.

Normalerweise erhält die *Fetch-Phase* die nächste *Order* aus der Eingabedatei. Es ist jedoch möglich, *Orders* der Eingabedatei voranzustellen. Diese *Orders* werden in der *Fetch-Phase* vor der nächsten *Order* der Eingabedatei gefunden. Dies geschieht mithilfe des *FetchStacks*, auf den *Orders* gelegt werden können. In der nächsten *Fetch-Phase* wird immer die oberste *Order* des *FetchStacks* geholt und daraufhin vom *FetchStack* entfernt. Die Eingabedatei befindet sich auf dem untersten Platz des *FetchStacks* und wird somit nur verwendet, wenn der Stack ansonsten komplett leer ist. Wann und wie *Orders* auf den *FetchStack* gelegt werden, wird im Abschnitt 4.3 erklärt. Die Kompilierung ist beendet, sobald der *FetchStack* leer ist.

4.2 Die *Enqueue-Phase*

Nachdem die nächste *Order* in der *Fetch-Phase* gefunden wurde, wird diese *Order* an die *Enqueue-Phase* weitergegeben. Während der *Enqueue-Phase* kommt die *OrderQueue* ins Spiel. Dabei handelt es sich, wie der Name schon sagt, um eine Queue von *Orders*. Die Aufgabe der *OrderQueue* ist das Speichern und spätere Zurückholen von *Orders*. Die *OrderQueue* kann mit *Instructions*, die im Abschnitt 4.3 weiter ausgeführt werden, aktiviert und deaktiviert werden. Wenn eine *Order* in die *Enqueue-Phase* gelangt und die *OrderQueue* aktiviert ist, wird diese *Order* der *OrderQueue* hinzugefügt. Die *Execute-Phase* wird danach übersprungen und der QHS-Zyklus beginnt von neuem bei Fetch. Die *Order* wurde (ohne die *Execute-Phase* erreicht zu haben) auf der *OrderQueue* gespeichert. Später ist es mit *Instructions* möglich diese *Order* von der *OrderQueue* zu entfernen und auszuführen.

Bestimmte *Orders* können jedoch *orderQueue-proof*, also immun gegen die *OrderQueue*, gemacht werden. *Orders*, die *orderQueue-proof* sind, werden an die *Execute-Phase* weitergegeben, auch wenn die *OrderQueue* aktiv ist. Dieses Prinzip ist zum Beispiel besonders bei der *Instruction*, welche die *OrderQueue* wieder deaktiviert, wichtig. Da diese *Instruction* ansonsten nicht zur *Execute-Phase* gelänge und somit die *OrderQueue* nie deaktiviert würde.

Ist die *OrderQueue* deaktiviert oder die *Order orderQueue-proof*, gelangt die *Order* zur *Execute-Phase*.

4.3 Die *Execute-Phase*

Während der *Execute-Phase* wird die Ausgabedatei generiert. Je nach Typ der *Order* (*Identifier*, *Instruction* oder *LiteralCode*) läuft die *Execute-Phase* sehr unterschiedlich ab. In den folgenden Abschnitten wird der Ablauf der *Execute-Phase* je nach *Ordertyp* erklärt.

4.3.1 *Identifier* in der *Execute-Phase*

Identifier stehen für die am Anfang von Kapitel 4 erwähnten Macros. Die Macros sind für den QHScompiler eine Liste an *Orders*. *Identifier* lassen sich zu einer Liste an *Orders* definieren. Wenn nun ein *Identifier* in die *Execute-Phase* gelangt, werden die dazugehörigen *Orders* auf den *FetchStack* aus Abschnitt 4.1 gelegt. Bei den nächsten *Fetch-Phasen* werden nun zuerst die zum *Identifier* gehörenden *Orders* nacheinander abgebaut. Einfach ausgedrückt wird der *Identifier* also durch seine *Orders* ersetzt. Als Beispiel seien folgende *Identifier* definiert:

Tabelle 4.2: Definition von *Identifiern* als Beispiel

Identifier	**Definition**
`id1`	`"lit1"`
`id2`	`#inst2.1 "lit2" #inst2.2`
`id3`	`id2 "lit3"`

Mit diesen *Identifiern*, wird nun folgender Code kompiliert:

Auflistung 4.1: QHS-Code zur Veranschaulichung von *Identifiern*

Eingabe

```
"lit0.1"
id1
"lit0.2" id2
id3
```

Ausgabe

```
"lit0.1"
"lit1"
"lit0.2" #inst2.1 "lit2" #inst2.2
#inst2.1 "lit2" #inst2.2 "lit3"
```

Die *Identifier* aus der Eingabe wurden mit ihren Definitionen ersetzt. Wie der *Identifier* `id3` zeigt, ist es möglich *Identifier* ineinander zu verschachteln.

Die Definitionen der *Identifier* sind in einem sogenannten *Environment* definiert. Bei einem *Environment* handelt es sich um eine einfache Map, die einen *Identifier* mit einer Liste an *Orders* verknüpft. *Environments* sind in einer verketteten Liste gespeichert. Neue *Environments* können dieser Liste hinzugefügt und aus der Liste entfernt werden. Das letzte *Environment* der Liste ist das älteste und das erste *Environment*

das neuste. Ein neuer *Identifier* wird immer zum ersten *Environment* der Liste hinzugefügt. Definitionen des gleichen *Identifiers* in älteren *Environments* werden nicht überschrieben oder gelöscht. Bei der Abfrage nach einem *Identifier* wird immer die neuste vorhandene Definition zurückgegeben. In Abschnitt 4.4.2 werden *Environments* anhand eines Beispiels erneut aufgegriffen.

4.3.2 *Instruction* in der *Execute-Phase*

Wenn eine *Instruction* in die *Execute-Phase* gelangt, wird die dazu definierte Funktion im QHScompiler ausgeführt. Diese Funktionen können *Identifier* definieren, die *OrderQueue* aktivieren, Zahlen addieren und vieles mehr. *Instructions* sind somit der Weg wie während der Kompilierung auf den QHScompiler Einfluss genommen werden kann. In der Tabelle 4.3 sind *Instructions* aufgelistet, die in späteren Beispielen verwendet werden.

Tabelle 4.3: Wichtige Instructions des QHScompilers

Instruction	**Beschreibung**
`#enterOrderQueue`	Aktiviert die *OrderQueue*. Diese *Instruction* ist *orderQueue-proof*.
`#exitOrderQueue`	Deaktiviert die *OrderQueue*. Diese *Instruction* ist *orderQueue-proof*.
`#assign`	Die erste *Order* der *OrderQueue* muss ein *Identifier* sein. Der Rest der *Orders* auf der *OrderQueue* wird als Definition für diesen *Identifier* festgelegt.
`#assignToOne`	Wie `#assign`, jedoch wird nach dem *Identifier* nur eine weitere *Order* von der *OrderQueue* genommen und als Definition für den *Identifier* verwendet.
`#force`	Die nächste *Order* wird nach der *Fetch-Phase* sofort an die *Execute-Phase* weitergegeben. Überspringt die *Enqueue-Phase* und somit die *OrderQueue*. Diese *Instruction* ist *orderQueue-proof*.
`#orderEnqueue`	Die nächste *Order* wird sofort dem Ende der *OrderQueue* hinzugefügt, auch wenn diese *Order orderQueue-proof* wäre. Die *Execute-Phase* wird übersprungen.
`#orderFrontEnqueue`	Ähnlich wie `#orderEnqueue`. Die *Order* wird jedoch an den ersten Platz der *OrderQueue* gesetzt.
`#deepFetch`	Die nächste *Order* der Eingabedatei wird oben auf den *FetchStack* gesetzt. Ermöglicht den Zugriff auf die Eingabedatei innerhalb eines *Identifiers*.
`#queueFetch`	Die erste *Order* der *OrderQueue* wird oben auf den *FetchStack* gesetzt.
`#pushEnv`	Ein neues *Environment* wird der *Environment-Liste* hinzugefügt.
`#popEnv`	Das neuste *Environment* der *Environment-Liste* wird gelöscht.
`#addToIdentifier`	Die erste *Order* der *OrderQueue* muss ein *Identifier* sein und die zweite *LiteralCode*. Die Definition des *Identifiers* muss eine Zahl sein. Der *LiteralCode* wird zu dieser Zahl addiert und im *Identifier* gespeichert.

Der QHScompiler umfasst 31 *Instructions*, wobei 5 dieser ausschliesslich fürs Debuggen des Compilers dienen.

4.3.3 *LiteralCode* in der *Execute-Phase*

LiteralCode ist der Weg wie der QHScompiler Assembly-Code generiert. Dieser ist sehr einfach. Wenn *LiteralCode* in die *Execute-Phase* gelangt, wird alles, das zwischen den Satzzeichen steht, in die Ausgabedatei geschrieben. Dies ist die einzige Möglichkeit des QHScompiler Assembly-Code zu generieren. Einzig die *LiteralCode-Orders* bestimmen, was in die Ausgabedatei gelangt, und somit welcher Sprache diese Ausgabedatei folgt. Durch das Anpassen der *LiteralCode-Orders* ist es also möglich die Ausgabesprache des QHScompilers zu ändern.

4.4 Definition der Macros

Somit ist der QHScompiler komplett. Grundsätzlich lässt sich mit *LiteralCode* bereits jedes Programm schreiben und mit dem QHScompiler kompilieren. Jedoch muss, wie in Kapitel 2 festgelegt, ie Eingabesprache eine C-ähnliche Syntax aufweisen und die Ausgabesprache Assembly sein. Um dies zu ermöglichen, muss man, wie zu Beginn von Kapitel 4 erwähnt, bestimmte Macros, also *Identifier*, definieren. In diesem Abschnitt werde ich zeigen, wie sich diese *Identifier* auch für syntaktisch komplexe Programmiersprachen definieren lassen. Zur Veranschaulichung dienen Variablen und Funktionsdefinitionen.

4.4.1 Die QHS-Notation

In diesem Abschnitt wird viel QHS-Code als Beispiel verwendet. Um die Leserlichkeit von QHS zu verbessern, werden zuerst paar *Identifier* anstelle der umständlichen *Instructions* definiert. Diese *Identifier* sind in der folgenden Tabelle 4.4 aufgeführt.

Tabelle 4.4: *Identifier* als Abkürzung von *Instructions*

Identifier	**Definition**
`[`	`#enterOrderQueue`
`]`	`#exitOrderQueue`
`>>`	`#assign`
`->`	`#assignToOne`
`!`	`#force`
`\n`	Eine neue Zeile in der Ausgabedatei
`/*`	Beginn eines Kommentars
`*/`	Ende eines Kommentars

Innerhalb von Kommentaren wird Pseudo-Code verwendet, um den QHS-Code verständlicher zu erklären. Kommentare können mehrere Zeilen umfassen und beginnen immer mit `/*` und enden mit `*/`. Der Kommentar `/* X = "test" #pushEnv */` würde bedeuten, dass der *Identifier* `X` als Folge der *Orders* `"test"` (*LiteralCode*) und `#pushEnv` (*Instruction*) definiert wurde.

4.4.2 Beispiele zu *Identifier*-Definitionen

Wie die Definition von *Identifiern* ablaufen kann, wird in diesem Abschnitt an einigen Beispielen erklärt.

Auflistung 4.2: Beispiel zu gewöhnlichen *Identifier*-Definitionen

```
1  ____________________ Eingabe ____________________
2  [ id1 "lit1" ] ->
3  [ id2 "lit2.1" "lit2.2" id1 ] >>
4  [ id3 ] [ "lit3.1" "lit3.2" ] >>
5
6  id1 \n
7  id2 \n
8  id3
9
10
11 ____________________ Ausgabe ____________________
12
13 lit1
14 lit2.1 lit2.2 lit1
15 lit3.1 lit3.2
```

Die Linien 2 und 3 definieren die folgenden beiden *Identifier*:

```
id1 = "lit1"
id2 = "lit2.1" "lit2.2" id1
```

In Linie 4 wird die *OrderQueue* zwischendurch deaktiviert und wieder aktiviert. Dies hat keinen Einfluss auf die Kompilierung und wird daher für die Verbesserung der Leserlichkeit von langen *Identifier*-Definitionen verwendet. Die *OrderQueue* sieht vor dem >> auf Linie 4 wie folgt aus, wobei die linkste *Order* die erste *Order* der *OrderQueue* darstellt:

```
id3 "lit3.1" "lit3.2"
```

Wie gewohnt wird `id3` mit >> definiert:

```
id3 = "lit3.1" "lit3.2"
```

Auflistung 4.3: Beispiel zu `#assignToOne`

```
1  ____________________ Eingabe ____________________
2  [ id4 "lit4" ] >>
3  [ id5 "lit5" id6 ] ->
4  [ "lit6" ] >>
5
6  id4 \n
7  id5 \n
8  id6
9
10
11
12 ____________________ Ausgabe ____________________
13
14 lit4
15 lit5
16 lit6
```

Die Definition von `id4` ist hier ähnlich wie in Linie 2 der Auflistung 4.2. Jedoch wird hier anstelle von -> (`#assignToOne`) der *Identifier* >> (`#assign`) verwendet. Da sich in beiden Fällen jedoch nur eine weitere *Order* neben dem *Identifier* auf der *OrderQueue* befinden, macht die Verwendung von >> keinen Unterschied. Bei Linie 3 und 4 ist dies jedoch anders. Zuerst werden drei *Orders* auf die *OrderQueue* gelegt. Der *Identifier* -> definiert daraufhin `id5` wie folgt:

```
id5 = "lit5"
```

Der *Identifier* `id6` bleibt auf der *OrderQueue*. Erst nachdem `"lit6"` der *OrderQueue* hinzugefügt wurde, wird `id6` durch >> definiert:

```
id6 = "lit6"
```

Auflistung 4.4: Beispiel zu `#orderFrontEnqueue`

```
1  ____________________ Eingabe ____________________
2  [ "lit7.1" ]
3  #orderFrontEnqueue id7
4  [ "lit7.2" ] >>
5
6  id7
7
8
9
10 ____________________ Ausgabe ____________________
11
12 lit7.1 lit7.2
```

Zuerst gelangt `"lit7.1"` auf die *OrderQueue*. Daraufhin wird mit der *Instruction* `#orderFrontEnqueue` der *Identifier* `id7` auf die erste Position der *OrderQueue* gesetzt. Die *OrderQueue* sieht daraufhin wie folgt aus:

```
id7 "lit7.1"
```

Der *LiteralCode* `"lit7.2"` wird normal hinten an die *OrderQueue* angefügt. Da sich der Identifier `id7` auf dem ersten Platz der *OrderQueue* befinden, kann dieser definiert werden:

```
id7 = "lit7.1" "lit7.2"
```

Auflistung 4.5: Beispiel zu doppelter Aktivierung der *OrderQueue*

```
1  ____________________ Eingabe ____________________
2  [ id8 ]
3  [
4      "lit8.1"
5      [ "lit8.2" ]
6  ] >>
7
8  id8
9
10
11
12 ____________________ Ausgabe ____________________
13
14 lit8.1
15 Die OrderQueue enthält: "lit8.2"
```

In Auflistung 4.5 wird eine *Identifier*-Definition auf mehrere Zeilen aufgeteilt. Dies dient der besseren Leserlichkeit und kompiliert gleich wie eine Definition auf nur einer Linie. Auf Linie 5 werden `[` (`#enterOrderQueue`) und `]` (`#exitOrderQueue`) innerhalb einer bereits aktiven *OrderQueue* verwendet. Im QHScompiler ist dies so implementiert, dass die beiden *Identifier* nicht ausgeführt und normal der *OrderQueue* hinzugefügt werden. Die *OrderQueue* enthält nach Linie 5 folgende *Orders*:

```
id8 "lit8.1" [ "lit8.2" ]
```

Die *Identifier* `[` und `]` gelangen normal in die *OrderQueue* und damit auch in die Definition von `id8`:

```
id8 = "lit8.1" [ "lit8.2" ]
```

Auflistung 4.6: Beispiel zu `#force`

```
1  ____________________ Eingabe ____________________
2  [ id9 "lit9" ] >>
3
4  [ id10 ]
5  [
6      ! id9
7      [ ! id9 ]
8  ] >>
9
10 id9  \n
11 id10
12
13
14
15
16 ____________________ Ausgabe ____________________
17
18 lit9
19 lit9
20 Die OrderQueue enthält: "lit9"
```

Zuerst wird in Linie 2 der *Identifier* `id9` wie gewohnt definiert. In Linie 6 wird der *Identifier* `!` (`#force`) verwendet. Die *Instruction* `#force` ist *orderQueue-proof* und zwingt den QHScompiler dazu, die nächste *Order* an die *Execute-Phase* weiterzugeben, obwohl die *OrderQueue* aktiv ist. Der *Identifier* `id9` wird daher mit seiner Definition ersetzt. Die *OrderQueue* sieht nach Linie 6 wie folgt aus:

```
id10 "lit9"
```

In Linie 7 wird daraufhin der *Identifier* `!` erneut verwendet. Dieses Mal jedoch innerhalb einer doppelten Aktivierung der *OrderQueue*, wie in Beispiel 4.5. In diesem Fall gelangen auch *Orders* die *orderQueue-proof* sind, nicht zur *Execute-Phase*. Der *Identifier* `!` wird normal der *OrderQueue* hinzugefügt. Die folgende Definition von `id10` lautet:

```
id10 = "lit9" [ ! id9 ]
```

Auflistung 4.7: Beispiel zu *Environments*

```
1  ____________________ Eingabe ____________________
2  [ id11 "lit11.1" ] >>
3  [ id12 "lit12" ] >>
4
5  id11 " " id12 \n
6
7  #pushEnv
8
9  [ id11 "lit11.2" ] >>
10 id11 " " id12 \n
11
12 #popEnv
13
14 id11 " " id12
15 ____________________ Ausgabe ____________________
16 lit11.1 lit12
17 lit11.2 lit12
18 lit11.1 lit12
```

Die *Identifier* `id11` und `id12` werden gewöhnlich definiert und verwendet. Daraufhin wird mit `#pushEnv` ein neues *Environment* hinzugefügt. In diesem *Environment* wird `id11` umdefiniert. In Linie 10 wird die neuste Definition der beiden *Identifier* verwendet. Für `id11` ist dies:

`id11 = "lit11.2"`

Der Definition des `id12` *Identifiers* ist unverändert:

`id12 = "lit12"`

Mit der *Instruction* `#popEnv` wird das neuste *Environment* gelöscht und die geänderte Definition von `id11` vergessen. Die Definition vom *Identifier* `id11` ist wieder:

`id11 = "lit11.1"`

4.4.3 Parameter und Rückgabewert für *Identifier*

Mit den `#enterOrderQueue` (resp. `[ )` und `#exitOrderQueue` (resp. `] )` *Instructions* kann innerhalb eines *Identifiers* die *OrderQueue* verwendet werden. Dies ermöglicht eine Art von Parameter und Rückgabewert für *Identifier*. Vor dem Aufruf eines *Identifiers* lassen sich *Orders* der *OrderQueue* hinzugefügen. Diese *Orders* können dann innerhalb des *Identifiers* wie Parameter verwendet werden. Genauso kann der *Identifier* *Orders* der *OrderQueue* hinzufügen und diese somit zurückgeben.

Auflistung 4.8: Verwendung von Parametern und Rückgabewert eines *Identifiers*

```
_____________________________________ Eingabe _____________________________________
[ foo ]
[
    #orderFrontEnqueue param1 ->    /* param1 = erstes Argument */
    #orderFrontEnqueue param2 ->    /* param2 = zweites Argument */

    param1 " : " param2 \n          /* param1 + " : " + param2 + "\n" */

    [ "Rückgabewert" ]              /* "Rückgabewert" wird der OrderQueue hinzugefügt */
] >>

[ "Argument 1" "Argument 2" ]       /* 2 Argumente werden der OrderQueue hinzugefügt */
foo                                 /* foo wird ausgeführt */
#queueFetch                         /* Die zurückgegebene Order wird von der OrderQueue
                                    geholt und ausgeführt */


_____________________________________ Ausgabe _____________________________________
Argument 1 : Argument 2
Rückgabewert
```

4.4.4 Variablen

Nun da die Grundkonzepte von QHS erklärt sind, geht es darum *Identifier* für einen C ähnlichen Syntax zu definieren. Die Umsetzung von Variablen in QHS ist einfach. Um Platz für die Variable auf dem Stack zu schaffen, muss zuerst die Grösse der Variable (für dieses Beispiel vier Bytes) vom *Stack-Pointer* subtrahiert werden. Dann wird für die Variable ein *Identifier* definiert, der zur Position der Variable auf dem Stack zeigt. Mit *LiteralCode* lässt sich dies wie folgt in QHS ausdrücken:

Auflistung 4.9: Definition einer Variable mit *LiteralCode*

Eingabe

```
"sub rsp, 4" \n
[ a "[rbp-4]" ] >>      /* a = "[rbp-4]" */

"add " a ", 5"
```

Ausgabe

```
sub rsp, 4
add [rbp-4], 5
```

Jedoch braucht man für diese Implementation immer noch Assembly Kenntnisse. Um die Definition von Variablen C ähnlicher zu machen, lässt sich zum Beispiel ein `var` *Identifier* definieren. Dieser `var` *Identifier* nimmt die Grösse der Variable als Argument über die *OrderQueue* an. Um die in C geläufige Syntax der Definition einer Variable beizubehalten, wird der Name der Variable mit der `#deepFetch` *Instruction* beschafft.

Auflistung 4.10: Definition einer Variable mit `var` *Identifier*

Eingabe

```
[ var ]
[
    #orderFrontEnqueue size ->          /* size = erstes Argument */
    [ name ! #deepFetch ] >>            /* name = Was nach dem var Identifier folgt */

    "sub rsp, " size \n

    [ ! name ] [ "[rbp-4]" ] >>             /* Was name enthält = "[rbp-4]" */
] >>

[ "4" ] var a
[ "8" ] var b

"add " a ", 5"
"sub " b ", 10"
```

Ausgabe

```
sub rsp, 4
sub rsp, 8
add [rbp-4], 5
sub [rbp-4], 10
```

Momentan erhält jede Variable jedoch noch die Adresse `rbp-4`, weswegen sich die Variablen gegenseitig überschreiben würden. Der momentane Abstand zum Base-Pointer muss also gespeichert und erhöht werden. Dafür wird bereits am Anfang des Programms ein *Identifier* `rbpOffset` als 0 definiert. Mit der `#addToIdentifier` *Instruction*, lässt sich nun `rbpOffset` erhöhen. Dies kann folgendermassen aussehen:

Auflistung 4.11: Definition einer Variable mit rbpOffset

Eingabe

```
[ rbpOffset "0" ] >>                    /* rbpOffset = "0" */

[ var ]
[
    #orderFrontEnqueue size ->          /* size = erstes Argument */
    [ name ! #deepFetch ] >>            /* name = Was nach dem var Identifier folgt */

    "sub rsp, " size \n

    [ rbpOffset ! size ] #addToIdentifier      /* rbpOffset += size */

    [ ! name ] [ "[rbp-" ! rbpOffset "]" ] >>  /* Was name enthält = "[rbp-OFFSET]" */
] >>

[ "4" ] var a
[ "8" ] var b

"add " a ", 5"
"sub " b ", 10"
```

Ausgabe

```
sub rsp, 4
sub rsp, 8
add [rbp-4], 5
sub [rbp-12], 10
```

Zuletzt lässt sich das umständliche Hinzufügen der Grösse der Variable sowie der `var` *Identifier* unter einem neuen *Identifier* zusammenfassen. Dies ist passenderweise die bekannte Bezeichnung für den Typen der Variable.

Auflistung 4.12: Definition einer Variable mit `int` *Identifier*

Eingabe

```
(...)

[ int ]
[
    [ "4" ] var
] >>

int a
int b

"add " a ", 5"
"sub " b ", 10"
```

Ausgabe

```
sub rsp, 4
sub rsp, 8
add [rbp-4], 5
sub [rbp-12], 10
```

Nun sieht die Definition und Verwendung einer Variable genau so aus, wie es in C gebräuchlich ist. Auf das Setzten von Variablen werde ich hier nicht weiter eingehen, da dies mit den Methoden, die im nächsten Abschnitt 4.4.5 erklärt werden, funktioniert.

4.4.5 Funktionsdefinitionen

Funktionen sind im Vergleich zu Variablen komplizierter. Nachfolgend sollen zwei der Probleme von Funktionsdefinitionen behandelt werden. Anhand einer Funktionsdefinition, wie sie zum Schluss aussehen sollte, will ich die beiden Probleme erläutern:

Auflistung 4.13: Ziel für die Definition einer Funktion in QHS

```
int foo ( int param1 , int param2 )
{
    (...)
}
```

Hier lässt sich bereits das erste Problem feststellen. Im vorherigen Abschnitt 4.4.4 wurde der `int` *Identifier* für die Definition einer Variable verwendet. Das `int` in der Auflistung 4.13 würde daher vom QHScompiler als Definition für eine Variable verstanden werden. Der Unterschied zwischen Variablen- und Funktionsdefinition besteht hierbei in den Klammern, die auf den Namen folgen. Der QHScompiler müsste also beim `int` *Identifier* nach vorne schauen, ob sich eine Klammer nach dem Namen befindet, und folglich eine Variablen- oder Funktionsdefinition ausführen. Bei einem traditionellen Compiler würde diese Überprüfung während der syntaktischen Analyse ausgeführt werden. Dies ist dem QHScompiler jedoch nicht möglich, da dieser, wie zu Beginn von Kapitel 4 erläutert, über keine syntaktische Analyse verfügt. Glücklicherweise lässt sich dieses erste Problem lösen, ohne eine Änderung am QHScompiler vorzunehmen. Die Lösung basiert darauf, beim `int` *Identifier* sowohl eine Variablen- als auch eine Funktionsdefinition vorzubereiten, aber keine der beiden bereits auszuführen. Daraufhin werden zwei *Identifier* definiert, erstens eine Klammer für eine Funktionsdefinition und zweitens ein Semikolon für die Definition einer Variable. Befindet sich nach dem Namen eine Klammer, wird eine Funktionsdefinition ausgeführt, ist dort aber ein Semikolon wird eine Variable definiert. Dieses Konzept wird im weiteren als *DelayedExecute* bezeichnet. Das Ganze sieht danach wie folgt aus:

Auflistung 4.14: Implementation eines *DelayedExecute* für Definitionen

```
(Definition einer Variable aus Auflistung 4.11)

[ function ]
[
    #orderFrontEnqueue returnSize ->        /* size = erstes Argument */
    #orderFrontEnqueue name ->              /* name = zweites Argument */

    [ ! name ] #orderToLiteral ":" \n       /* "foo:" */
] >>

[ definition ]
[
    #orderFrontEnqueue size ->          /* size = erstes Argument */
    [ name ! #deepFetch ] >>            /* name = Was nach dem var Identifier folgt */

    [ ; ]
    [
        [ ! size ! name ] var
    ] >>
    /* ; = [ size name ] var */

    [ ( ]
    [
        [ ! size ! name ] function
    ] >>
    /* ( = [ size name ] function */

] >>

[ int ]
[
    [ "4" ] definition
] >>
```

Das zweite Problem sind die Parameter einer Funktionsdefinition. Diese sehen genau gleich aus wie die Definition einer Variable, sollten jedoch vom QHScompiler anders ausgeführt werden. Erstens wird bei der Definition von Variablen aus Abschnitt 4.4.4 *LiteralCode* zur Subtraktion vom *Stack-Pointer* verwendet. Diese Subtraktion braucht es bei Parameterdefinition nicht. Zweitens verwendet eine Parameterdefinition einen anderen rbp-Offset als Definitionen von Variablen. Die Lösung liegt im Umdefinieren des `definition` *Identifiers*. Dieser ist momentan für die Definition von Variablen und Funktionen verantwortlich. Bei der Anfangsklammer der Funktionsdefinition wird der `definition` *Identifier* neu definiert, sodass er eine Parameterdefinition ausführt. Die vorherige Definition geht dank der `#pushEnv` *Instruction* nicht verloren. Bei der schliessenden Klammer wird `#popEnv` durchgeführt, und der `definition` *Identifier* ist wieder für Variablen und Funktionen zuständig. Diese Lösung wird im folgenden *TempAssign* genannt. Dies lässt sich in QHS wie folgt umsetzen:

Auflistung 4.15: Implementation eines *TempAssigns* für Parameter Definitionen

```
[ function ]
[
    #pushEnv

    #orderFrontEnqueue returnSize ->        /* size = erstes Argument */
    #orderFrontEnqueue name ->              /* name = zweites Argument */

    [ ! name ] #orderToLiteral ":" \n       /* "foo:" */

    [ definition paramDefinition ]          /* definition = paramDefinition */

    [ ) ] [ #popEnv ] >>                    /* ) = #popEnv */
] >>
```

Der *Identifier* `paramDefinition` ist ähnlich wie der `var` *Identifier* aus Abschnitt 4.4.4. Es wird bloss anstelle von `rbpOffset` ein neuer `paramOffset` *Identifier* verwendet und der Assembly-Code fürs Subtrahieren vom *Stack-Pointer* nicht hinzugefügt.

Nun fehlt nur noch etwas an der Funktionsdefinition: Der Funktionsbody. Dieser ist vergleichsweise einfach. Die beiden geschwungenen Klammern werden zu einem leeren *Identifier* definiert und somit ignoriert. Der gesamte Code innerhalb des Body wird ganz normal vom QHScompiler ausgeführt und an die Ausgabedatei angehängt. Das Endresultat sieht wie folgt aus:

Auflistung 4.16: Finale Definition einer Funktion in QHS

Eingabe

```
int foo ( int param1 , int param2 )
{
    "add " param1 ", " param2
}
```

Ausgabe

```
foo:
add [rbp+16], [rbp+20]
```

Wie das Beispiel der Funktionsdefinition zeigt lassen sich mit *DelayedExecute* und *TempAssign* auch syntaktisch komplexe Programmiersprachen in QHS definieren und mit dem QHScompiler kompilieren.

Um einen C ähnlichen Syntax zu ermöglichen, braucht der QHScompiler noch viele weitere *Identifier*. Da diese jedoch einem ähnlichen Prinzip wie die beschriebenen Definitionen von Variablen und Funktionen folgen, werde ich sie hier nicht weiter betrachten.

5 Auswertung des Compiler Vergleichs

Abschliessend will ich den QHScompiler, wie in Kapitel 2 beschrieben, mit zwei weiteren Compilern vergleichen. Bewertet werden die Compiler in Geschwindigkeit der Kompilierung, Geschwindigkeit der Ausgabedatei, Umgang mit fehlerhaftem Code und Offenheit für Erweiterung.

5.1 Geschwindigkeit der Kompilierung

Für die Messung der Kompilierungsdauer wird eine Funktion, die prüft, ob eine Zahl eine Primzahl ist, kompiliert. Diese Funktion wurde so geschrieben, dass jedes Feature, das alle drei Compiler unterstützen, verwendet wird. Dazu gehören Variablen, Funktionen und *Expressions* sowie Verzweigungen und Schleifen. Die Funktion wurde in die jeweiligen Sprachen übersetzt und mehrmals in ein Programm eingefügt. Anschliessend wurde jedes Programm zehnmal kompiliert. Die durchschnittliche Dauer der Kompilierung nach Menge an Kopien der Funktion in der Eingabedatei ist in Abbildung 5.1 ersichtlich.

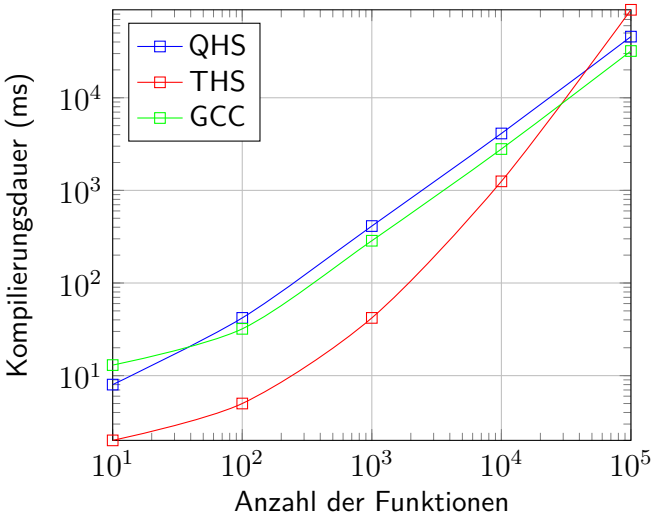

Abbildung 5.1: Vergleich der Kompilierungsdauer mit logarithmischen Skalen

Der THScompiler glänzt bei einer kleinen Programmgrösse mit einer sehr schnellen Kompilierung, doch steigt seine Kompilierungsdauer mit einer zunehmenden Grösse der Eingabedatei exponentiell an. Ab hunderttausend Funktionen kompiliert der THScompiler länger, als die beiden anderen Compiler. Bei noch grösseren Eingabedateien ist zu erwarten, dass der THScompiler weiterhin langsamer kompiliert als GCC und der QHScompiler. Für die exponentielle Kompilerungsdauer des THScompilers habe ich leider keine Erklärung. Theoretisch sollten alle Schritte, die der THScompiler durchläuft, eine lineare Komplexität aufweisen. Bei wenigen Kopien der Funktion kompiliert der THScompiler deutlich schneller als GCC, obwohl beide Compiler dem traditionellen Aufbau folgen. Dies liegt wahrscheinlich daran, dass der THScompiler

über weniger Funktionalitäten als GCC verfügt. Der QHScompiler, der einen ähnlichen Umfang wie der THScompiler umfasst, wird für kleine Eingabedateien klar vom THScompiler geschlagen.

Sowohl der QHScomiler als auch GCC beginnen mit einer hohen Kompilierungsdauer für eine tiefen Anzahl an Funktionen, verhalten sich später aber linear. Der Unterschied zwischen den Kompilierungsdauern von GCC und dem QHScompiler erscheint durch die logarithmischen Skalen konstant. Tatsächlich braucht der QHScompiler aber ab einer Programmgrösse über hundert Funktionskopien ungefähr 1,5-mal länger als GCC.

Somit schneidet sowohl bei kleinen als auch bei grossen Eingabedateien der traditionelle Compileraufbau besser ab, als mein alternatives Modell.

5.2 Geschwindigkeit der Ausgabedatei

Die Geschwindigkeit eines kompilierten Programmes wird anhand eines Algorithmus zur Berechnung von Primzahlen gemessen. Wie bei der Funktion aus Abschnitt 5.1 ist dieser Algorithmus so geschrieben, dass er möglichst jedes von allen drei Compilern unterstütze Feature verwendet. Der Algorithmus wurde für verschiedene Mengen an zu berechnenden Primzahlen je zehnmal ausgeführt und die Ausführungsdauer gemessen. In der folgenden Abbildung 5.2 ist die durchschnittliche Ausführungsdauer der Programme nach Menge an berechneten Primzahlen dargestellt.

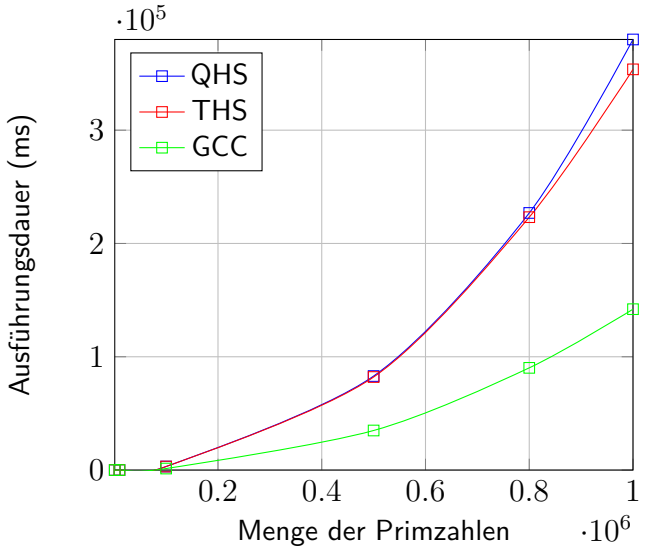

Abbildung 5.2: Vergleich der Ausführungsdauer mit GCC Optimierung

Wie in Abbildung 5.2 ersichtlich, beginnen alle drei kompilierten Programme mit einer sehr tiefen Ausführungsdauer. Die Programme des THS- und QHScompilers werden bis zum Schluss nahezu gleich schnell ausgeführt. Das von GCC generierte Programm ist jedoch deutlich schneller als die Programme des THS- und QHScompilers. Dies liegt ganz klar an den Optimierungsmethoden von GCC. Wenn man die Optimierung beim Kompilieren mit GCC deaktiviert, sieht die Abbildung wie folgt aus:

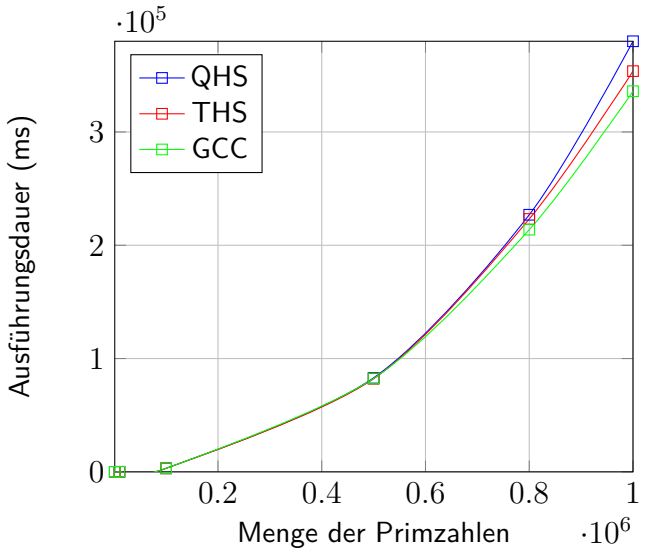

Abbildung 5.3: Vergleich der Ausführungsdauer ohne GCC Optimierung

Wie Abbildung 5.2 zeigt, wird das GCC Programm ohne Optimierung nur noch leicht schneller ausgeführt als die Programme der beiden anderen Compiler. Da ebenfalls weder der THS- noch der QHScompiler über Optimierungsmethoden verfügen, ist dies das erwartete Resultat.

Zusammengefasst lässt sich sagen, dass ohne Optimierung mein alternativer Aufbau eines Compilers ungefähr gleich schnelle Ausführungsgeschwindigkeit liefert, wie ein traditioneller Compiler. Trotzdem muss angemerkt werden, dass Optimierung bei einem traditionellen Compiler möglich ist und die Ausführungsdauer deutlich verringert, wie am Beispiel von GCC in Abbildung 5.2 gesehen werden kann. Ich kann mir vorstellen, dass Optimierung für einen nach meinem alternativen Aufbau entwickelten Compiler jedoch sehr schwierig wäre. Traditionelle Compiler haben die Möglichkeit Optimierungen auf Zwischencode auszuführen. Für meinem alternativen Compiler ist dies nicht möglich, da gar keine syntaktische Analyse durchgeführt und nie Zwischencode generiert wird. Auch steht die Grundidee meines alternativen Aufbaus der Optimierung stark im Weg. Wie zu Beginn von Kapitel 4 beschrieben, basiert mein Ansatz darauf, dass die von der *Macro Expansion* verwendeten Macros erst während der Kompilierung definiert werden. Daher kennt der QHScompiler vor der Kompilierung weder die Eingabe- noch die Ausgabesprache. Dies führt dazu, dass auch mögliche Optimierungsmethoden erst während dem Kompilieren gefunden werden können. Aus diesen Gründen wäre Optimierung für einen Compiler nach meinem alternativen Aufbau deutlich komplexer, wenn nicht sogar unmöglich.

5.3 Umgang mit fehlerhaftem Code

GCC und der THScompiler folgen beide einer exakt definierten Syntax und einer klaren Semantik. Wird die Syntax oder Semantik nicht eingehalten wird ein Fehler gemeldet. Dies ist ein Resultat der syntaktischen und semantischen Analyse, die nach bestimmten Regeln geschrieben wurden und diese Regeln exakt einhalten. Traditionelle Compiler sind dadruch sehr genau und hilfreich beim Finden und Melden von Fehlern.

Der QHScompiler arbeitet hingegen nach viel ungenaueren Regeln. Wie im Abschnitt 4.4.5 bereits be-

schrieben, verfügt der QHScompiler über keine Möglichkeit zu überprüfen, ob eine bestimmte *Order* folgt oder nicht. Er führt konsequent nur aus, was als Nächstes auftaucht. Darum führt ein fehlendes Zeichen nicht immer zu Fehlern. Folgender Code soll dies veranschaulichen:

Auflistung 5.1: QHS mit fehlenden *Tokens*

```
int a = "69"    /* ; fehlt */
foo ( a  ;      /* ) fehlt */
```

Der Code aus Auflistung 5.1 lässt sich einwandfrei vom QHScompiler kompilieren und daraufhin ausführen. Weder das Fehlen des Semikolons noch der schliessenden Klammer meldet der QHScompiler als Fehler. Die resultierende Ausgabedatei ist ebenfalls fehlerfrei und lässt sich einwandfrei ausführen. Dies ist jedoch bei folgendem Beispiel nicht mehr der Fall.

Auflistung 5.2: QHS mit fehlender öffnender Klammer

```
int a = "69"    /* ; fehlt */
foo a ) ;       /* ( fehlt */
```

Der Code bei Auflistung 5.2 kompiliert problemlos, der generierte Assembly-Code ist jedoch fehlerhaft. Die Funktion `foo` wird nicht ausgeführt und die Variable a nicht als Argument erkannt. Es entsteht also eine fehlerhafte Ausgabedatei, ohne dass der QHScompiler dies meldet.

Führt die Kompilierung doch zu einer Fehlermeldung, ist diese nicht immer besonders verständlich.

Auflistung 5.3: QHS mit falscher Anzahl Argumente

Eingabe

```
void foo ( ) { }

start
{
    int a = "69"
    foo ( a ) ;

    exit ;
}
```

Ausgabe

```
[ERROR] Cannot dequeue, OrderQueue is empty!
[ERROR] Expected LiteralCode for #literalToIdentifier at OrderQueue second, got: NONE
[ERROR] Cannot dequeue, OrderQueue is empty!
[ERROR] Tried #changeIntVar but second order (change) from OrderQueue is not direct code
[ERROR] Expected LiteralCode for #literalToIdentifier, got: NONE
[ERROR] Expected LiteralCode for #literalToIdentifier, got: NONE
[ERROR] Expected LiteralCode for #literalToIdentifier, got: NONE
```

Bei Auflistung 5.3 wird die Funktion `foo` ohne Parameter definiert, später jedoch mit einem Argument aufgerufen. Der QHScompiler verfügt über keine Möglichkeit, die Menge an Argumenten zu überprüfen, und meldet nicht direkt einen Fehler. Sobald er jedoch versucht die Grösse des erwarteten Argumentes von der *OrderQueue* zu holen ist diese leer. Der QHScompiler meldet einen Fehler gefolgt von vielen Folgefehlern.

Somit ist der QHScompiler bei der Meldung von Fehlern einerseits nicht sehr streng, andererseits aber auch verwirrend und ungenau bei der Fehlermeldung. Deshalb komme ich zum Schluss, dass der traditionelle

Compileraufbau mit syntaktischer und semantischer Analyse deutlich besser mit Fehlern umgehen kann als mein alternatives Modell.

5.4 Offenheit für Erweiterung

Seit der ersten Veröffentlichung im Jahr 1987 wird GCC bis heute weiterentwickelt. Mittlerweile lässt sich mit der GNU Compiler Collection nicht nur C, sondern auch viele weitere Programmiersprachen kompilieren. Jedoch erfordert jede Erweiterung einen Eingriff in den Code des Compilers.

Mein alternativer Aufbau bietet hingegen Möglichkeiten zur Erweiterung, ohne den Quellcode des Compilers bearbeiten zu müssen. Wie in Kapitel 4 bereits angemerkt, werden die Ein- und Ausgabesprachen mittels *Identifiern* innerhalb der Eingabedatei definiert. Dies ermöglicht es bloss durchs Verändern der Eingabedatei die Eingabesprache zu erweitern oder innerhalb der Datei zu wechseln. Compiler die meinem alternativen Aufbau folgen, lassen sich also ohne jegliche Veränderung am Quellcode erweitern.

5.5 Fazit

Im Vergleich mit traditionellen Compilern zeigt der von mir entwickelte QHScompiler einige Schwächen. Er ist sowohl in der Geschwindigkeit der Kompilierung als auch bei der Ausführungsdauer eines kompilierten Programmes einem traditionellen Compiler unterlegen. Beim Umgang mit Fehlern ist der QHScompiler weniger streng aber auch deutlich unpräziser und verwirrender als Compiler nach dem traditionellen Aufbau. Als einziger Vorteil meines alternativen Aufbaus lässt sich seine Offenheit für Erweiterung sehen.

Mithilfe eines Profilers habe ich die Kompilierungsdauer des QHScompilers analysiert. Daraus schloss ich, dass das System der *Identifier* besonders ineffizient ist. Jeder *Identifier* benötigt zuerst eine Abfrage bei den *Environments*. Diese Abfrage ist an sich keine aufwendige Sache, jedoch sind *Identifier* häufig ineinander verschachtelt. Daher werden auch für die Kompilierung von wenig Code viele *Identifier* ausgeführt. Generell liesse sich die Implementation der *Identifier* sicherlich stark verbessern und der gesamte QHScompiler optimieren.

Aus dem Vergleich des Umgangs mit fehlerhaftem Code wird ausserdem klar, dass die syntaktische Analyse für die angenehme Verwendung eines Compilers äusserst wichtig ist. Durch den Parser lassen sich Fehler in der Eingabedatei früh finden und genau melden. Dem QHScompiler ist dies, aufgrund der fehlenden syntaktischen Analyse, nicht möglich.

Ausserdem ist ein AST, wie in Abschnitt 5.2 thematisiert, auch für die Optimierung der Ausgabedatei äusserst wichtig. Die Optimierungsmethoden des QHScompiler sind daher bereits durch die fehlende syntaktische Analyse eingeschränkt. Doch auch abgesehen vom AST ist Optimierung für meinen alternativen Compileraufbau äusserst schwierig. Da die Eingabesprache erst während der Kompilierung definiert wird, müssen auch passende Optimierungsmethoden spontan gefunden werden. Dies ist aufwendig und würde sich wahrscheinlich in einer hohen Kompilierungsdauer äussern. Alternativ könnte man Optimierungsmethoden ebenfalls in der Eingabedatei definieren. Wie sich dies genau umsetzen liesse, weiss ich jedoch nicht.

Der einzige Vorteil des QHScompilers liegt in der Offenheit für Erweiterung. Das Wechseln der Programmiersprache innerhalb einer Datei ist definitiv interessant, jedoch habe ich noch kein Beispiel gefunden, wofür dieser Wechsel nötig wäre. In den meisten Fällen könnte man auch die Teile mit unterschiedlichen Sprachen auf mehrere Dateien aufteilen, einzeln kompilieren und danach mit einem *Linker* kombinieren.

Zusammengefasst führen während der Kompilierung definierte *Identifier* zu hohen Kompilierungsdauern, ungenauem Umgang mit Fehlern und mangelhafter Optimierung der Ausgabedatei. Der QHScompiler ist einem traditionellen Compiler also deutlich unterlegen.

6 Schluss

Der QHScompiler schneidet im Vergleich zu einem traditionellen Compiler leider schlechter ab. Trotzdem habe ich mit meiner Arbeit das erreicht, was ich mir erhofft hatte. Der Auslöser für meinen alternativen Ansatz war die Frage: Wieso sind Compiler so, wie sie sind? Dies konnte ich für mich ganz klar beantworten. Die syntaktische Analyse, die ich zuerst für unnötig hielt, erfüllt eine wichtige Aufgabe für Fehlermeldung und Optimierung. Ausserdem ist man dank der syntaktischen Analyse nicht auf verwirrende Methoden wie *DelayedExecute* oder *TempAssign* aus Abschnitt 4.4.5 angewiesen. Das Definieren der Macros und somit der Eingabe- und Ausgabesprache während der Kompilierung klingt zuerst verlockend, bringt jedoch auch Probleme fürs Optimieren der Ausgabedatei mit sich. Compiler folgen also aus gutem Grund dem traditionellen Aufbau.

Es hat mir sehr viel Spass gemacht, meinen alternativen Ansatz zu entwickeln. Auch wenn es häufig frustrieren kann, ist es doch viel aufregender einer eigenen Idee zu folgen, als einfach stur dem traditionellen Weg zu folgen. Sowohl bei der Entwicklung des QHS- als auch des THScompiler folgte ich häufig schlechten Ideen und zu hohen Ansprüchen, bemerkte dies meist jedoch erst, als ich schon eine Woche an ihnen verloren hatte. Sehr hilfreich war es dabei die Entwicklung parallel bereits in Worte zu fassen. Die Verschriftlichung meiner Gedanken half mir dabei ein Verständnis aufzubauen, wie mein Programm tatsächlich funktioniert. Leider ist mir dies erst sehr spät aufgefallen.

Zuletzt möchte ich nochmals hervorheben, wie wichtig die Vorlesung zum Thema Compiler Construction der Universität Bern [2] für diese Maturaarbeit war. Ohne diese Vorlesung wäre es mir viel schwerer gefallen, den traditionellen Aufbau eines Compilers zu verstehen und umzusetzen. Meine Maturaarbeit wie sie jetzt steht, wäre ohne die Vorlesung wahrscheinlich nicht möglich gewesen.

Die beiden von mit geschriebenen Compiler sowie Einblick in deren Entwicklungsprozess lassen sich unter `https://github.com/TheScaryParrot/MyCompiler` finden.

Abbildungsverzeichnis

Auflistungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2006.

[2] Prof. Timo Kehrer der Universität Bern. *Vorlesung zu Compiler Construction*, 2024.

[3] Susan L. Graham. *Table-Driven Code Generation*. 1982. [Online; Zugriff am 7.9.2024].

[4] Bob Nystrom. *Crafting Interpreters: A Map of the Territory*. `https://github.com/munificent/craftinginterpreters/blob/master/site/image/a-map-of-the-territory/mountain.png` [Online; Zugriff am 8.10.2024].

[5] Wikipedia. *Abstract Syntax Tree*, `https://en.wikipedia.org/wiki/Abstract_syntax_tree` [Online; Zugriff am 8.10.2024].