

Maturaarbeit

Compiler Construction

Fabio Stalder

Betreut durch
Thomas Jampen

25. September 2024



Gymnasium Kirchenfeld
Abteilung MN

Inhaltsverzeichnis

1	Einleitung	4
2	Vergleich der Compiler	5
3	Funktionsweise traditioneller Compiler	6
3.1	Lexikalische Analyse	7
3.2	Syntaktische Analyse	7
3.3	Semantische Analyse	8
3.4	Codegenerierung	8
3.5	Optimierung	9
4	Der QHScompiler	10
4.1	Die Fetch-Phase	10
4.2	Die Validate-Phase	11
4.3	Die Execute-Phase	12
4.3.1	Identifizier	12
4.3.2	Instructions	12
4.3.3	LiteralCode	13
4.4	Definition der Macros	13
4.4.1	Abkürzungen	14
4.4.2	Identifizier Parameter und Rückgabewert	14
4.4.3	Variablen	15
4.4.4	Funktionsdefinitionen	17
5	Auswertung des Compiler Vergleichs	20
5.1	Geschwindigkeit der Kompilierung	20
5.2	Geschwindigkeit der Ausgabedatei	21
5.3	Umgang mit fehlerhaftem Code	22
5.4	Offenheit für Erweiterung	24
5.5	Fazit	24
6	Schluss	26

In folgendem Text werde ich die Idee, Entwicklung und schlussendliche Auswertung eines von mir erdachten alternativen Ansatzes für den Aufbau eines Compilers beschreiben.

1 Einleitung

In der Informatik beschreibt *Compiler* ein Programm, das Code aus einer Programmiersprache in eine andere Programmiersprache übersetzt. In dieser Hinsicht gleichen Compiler Übersetzern für Menschensprache. Genauso 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 verständlicheren Programmiersprachen zu Assembly und ermöglichen daher ein viel einfacheres Schreiben von Programmen. Compiler unterscheiden sich jedoch grundsätzlich von Übersetzern in der Erwartungshaltung, die an sie gestellt wird. Menschensprache ist sehr komplex und nicht immer besonders 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. Dieser sollte optimal generiert werden, um die schlussendliche Ausführungsdauer so kurz wie möglich zu halten. Sollte sich doch einmal ein Fehler in der Eingabedatei befinden, muss dieser verständlich gemeldet werden. 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 darin freundlicherweise durch eine Vorlesung der Universität Bern. Während mir in dieser Vorlesung der theoretische Aufbau eines Compilers gelehrt wurde, fragte ich mich trotzdem hin und wieder, wieso genau ein Compiler tatsächlich so aufgebaut ist 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" Compiler Aufbaus 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 RegEx, Datenstrukturen und Assembly vorausgesetzt.

2 Vergleich der Compiler

Um die Leistung meines alternativen Aufbaus eines Compilers zu testen, werde folgende drei Compiler verglichen:

1. Der *QHScompiler* ist der von mir nach meinem Aufbau entwickelte Compiler. Seine genaue Funktionsweise wird in Abschnitt 4 ausgeführt. Er ist in C++ geschrieben und verwendet meine eigene Programmiersprache QHS als Eingabesprache.
2. *GCC* ist der gebräuchlichste Compiler für die Programmiersprache C. Veröffentlicht im Jahre 1987 wird GCC bis heute weiterentwickelt und ermöglicht inzwischen auch die Kompilierung von C++, Rust, Fortran, usw. Passend dazu lautet das Akronym GCC ausgeschrieben: GNU Compiler Collection. Für diesen Vergleich repräsentiert GCC den traditionellen Compiler Aufbau.
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.

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

Tabelle 2.1: Anforderungen an die Compiler

Ausgabe als Assembly Code	Die Ausgabesprache muss Assembly Code sein
C ähnlicher Syntax	Die Eingabesprache muss einen C ähnlichen Syntax aufweisen
Variablen und Funktionen	Lokale und globale Variablen sowie Funktionen müssen unterstützt werden
Branching und Loops	If-Statements und einfache Loops müssen umsetzbar sein

Die Kriterien des Vergleichs sind in Tabelle 2.2 aufgelistet.

Tabelle 2.2: Vergleichskriterien der Compiler

Geschwindigkeit der Kompilierung	Wie lange dauert die Kompilierung von Code?
Geschwindigkeit der Ausgabedatei	Wie lange dauert die Ausführung der Ausgabedatei?
Umgang mit fehlerhaftem Code	Wie geht der Compiler mit fehlerhaften Eingabedateien 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:

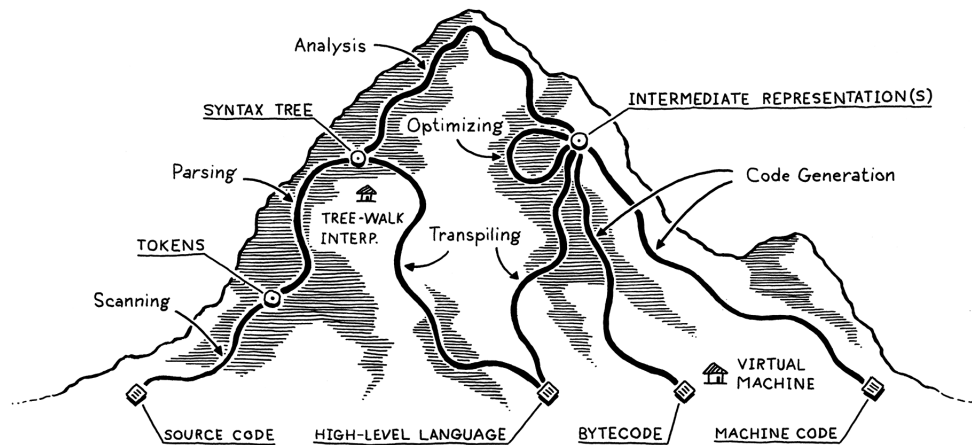


Abbildung 3.1: Schritte, die ein Compiler durchläuft

In dieser Arbeit werde ich mich auf die in der unteren Abbildung 3.2 dargestellten Schritte fokussieren.

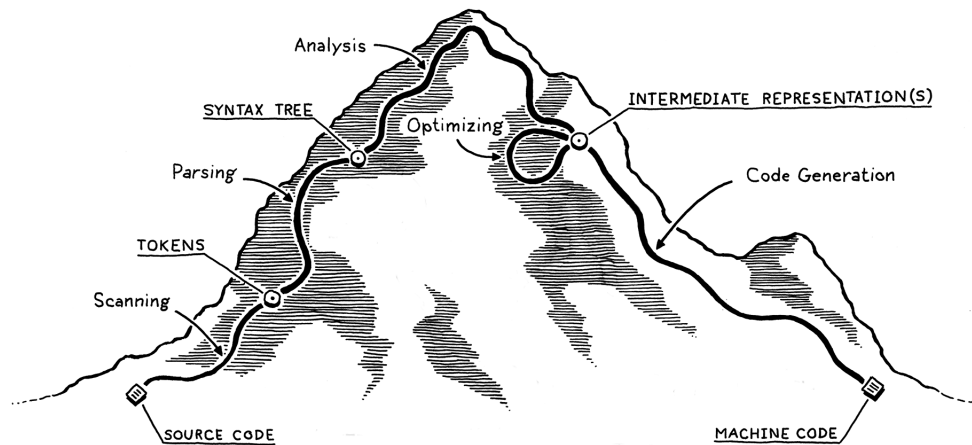


Abbildung 3.2: Schritte, die in dieser Arbeit behandelt werden

Die von mir verwendeten Fachbegriffe entsprechen hierbei nicht immer denen aus Abbildung 3.2.

Als Basis für dieses Kapitel dient grösstenteils eine Vorlesung der Universität Bern, die ich für freundlicher-weise besuchen durfte.

3.1 Lexikalische Analyse

Meist werden Programme so geschrieben, dass wir Menschen sie lesen und verstehen können. Dafür verwendet man Buchstaben, Zahlen, Sonderzeichen (wie + oder *) und Whitespaces (wie Leerzeichen oder Absätze). 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 dieses Lexers ist es, die Eingabedatei zu analysieren 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 auf der Programmiersprache C:

Auflistung 3.1: C code vor der lexikalischen Analyse

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

    return 1;
}
```

Auflistung 3.2: Tokens nach der lexikalische 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 Eingabe-Programmiersprache 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

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 Punktvorstrich-Regeln Expressions 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.

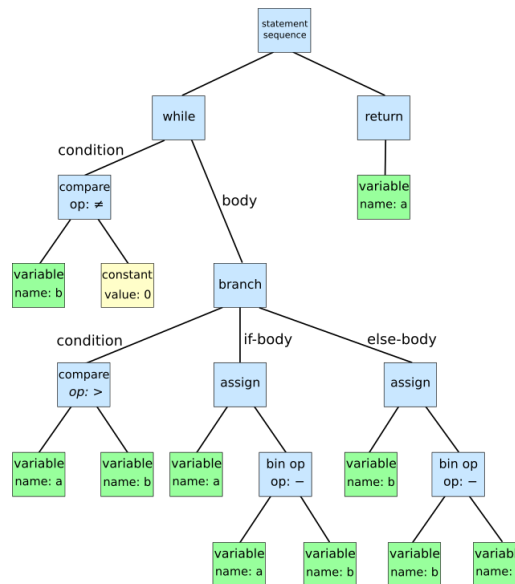


Abbildung 3.3: Abstract Syntax Tree zum Euklidischen Algorithmus

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 wie 'if' oder 'for' 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

Semantik ist die Wissenschaft der Bedeutung von Wörtern einer Menschensprache. Bei einem Compiler geht es bei der *semantische Analyse* nicht um 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 ein impliziter Wechsel des Datentyps, hinzugefügt werden. So geben zum Beispiel manche Programmiersprachen bei der Division zweier Integers eine Float zurück. Auch werden unbekannte Variablen und Funktionen in diesem Schritt abgefangen. Weiter wird der Datentyp einer Node des AST an diese angebunden.

3.4 Codegenerierung

Codegenerierung ist der finale und oft auch komplexeste Schritt, der ein Compiler ausführen muss. Nun da die Eingabedatei als AST vorliegt, kann die Ausgabedatei generiert werden. Eine geläufige Methode der Codegenerierung ist die sogenannte *Macro Expansion*. Hierbei wird der AST von unten nach oben schritt-

weise mit Teilen an Ausgabecode ersetzt. Diese Ausgabecode-Teile sind häufig von den darunterliegenden Nodes abhängig.

MAYBE FIGURE OR LISTING AS EXAMPLE

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 und dies häufig mehrmals. Dabei geht es darum die Ausgabedatei so effizient wie möglich zu machen. Effizient kann hierbei verschiedenes bedeuten. Die Ausgabedatei muss so schnell wie möglich ausgeführt werden, Memory 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 sogenannten Deadstores. Es muss von CPU-Registern profitiert, mit Heap-Memory umgegangen und von Inline-Funktionen Gebrauch gemacht werden. Optimierung ist also sehr vielseitig und komplex. Wie Optimierung genau aussehen kann, wird daher in dieser Maturaarbeit nicht weiter betrachtet.

4 Der QHScompiler

Der QHScompiler basiert auf einem von mir erdachten alternativen Aufbau für einen Compiler. Diesem Aufbau liegt eine einfache Idee zugrunde: Die Macros die der Macro Expansion aus Abschnitt 3.4 sollen erst während der Kompilierung 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 selbige 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, folgt mein alternativer Ansatz einem Aufbau, der sich stark von einem traditionellen Compiler unterscheidet.

Die Programmiersprache, in der sich meine Macros definieren lassen, bezeichne ich als QHS. Der Compiler der QHS versteht und kompiliert, nenne ich dazu passend den QHScompiler. 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 steht ein einfacher Zyklus zugrunde, dessen Vorbild der Von-Neumann Zyklus ist.

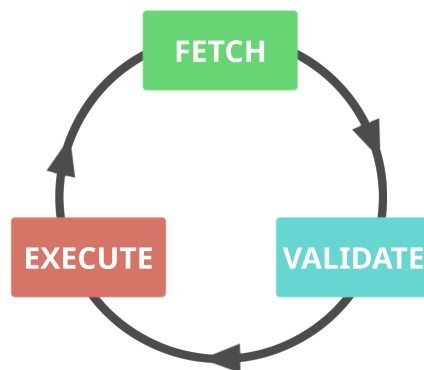


Abbildung 4.1: Zyklus der QHS Kompilierung (QHS-Zyklus)

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 lexikalische 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 RegEx definiert. Whitespaces dienen als Trennung zwischen Orders und werden ignoriert.

Tabelle 4.1: RegEx Definitionen der Ordertypen

identifier	<code><identiferChar>*</code>
instruction	<code># <identiferChar>*</code>
literalCode	<code>".*"</code>

`<identiferChar>` = `[^# "<whitespace>]`
`<whitespace>` = `SPACE | NEWLINE | TAB`

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 zuerst 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.

FETCHSTACK FIGURE

Die Hauptanwendung des *FetchStacks* wird im Abschnitt 4.3 erklärt. Die Kompilierung ist beendet, sobald keine Order mehr auf dem *FetchStack* vorhanden ist.

4.2 Die Validate-Phase

Nachdem die nächste Order in der Fetch-Phase gefunden wurde, wird diese Order an die Validate-Phase weitergegeben. Während der Validate-Phase kommt die *OrderQueue* ins Spiel. Dabei handelt es sich, wie der Name schon sagt, um eine Queue an 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 Validate-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. Zu beachten ist, dass LiteralCode nicht orderQueue-proof sein kann.

MAYBE CODESTACK FIGURE

Ist die OrderQueue deaktiviert oder die Order orderQueue-proof, wird diese Order an den letzten Schritt Execute weitergegeben.

4.3 Die Execute-Phase

Während der Execute-Phase wird der tatsächliche Assembly Code 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

Identifier stehen für die am Anfang von Abschnitt 4 erwähnten Macros. Diese Identifier sind in einem *Environment* definiert. Bei einem Environment handelt es sich um eine einfache Map, die einen Identifier mit einer Liste an Orders verknüpft. 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 mit seinen Orders ersetzt.

EXAMPLE FIGURE?

Die bereits erwähnten Environments sind dabei in einer Linked-List gespeichert. Somit können neue Environments zu dieser Liste hinzugefügt und von der Liste entfernt werden. Das unterste Environment der Liste ist das älteste und das oberste Environment das neuste. Ein neuer Identifier wird immer zum obersten Environment 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. Ist keine vorhanden, wird ein Error ausgegeben.

4.3.2 Instructions

Wenn eine Instruction in die Execute-Phase gelangt, wird die dazu definierte Funktion im QHScompiler ausgeführt. Diese Funktionen können Variablen im QHScompiler speichern, die OrderQueue aktivieren, Identifier definieren und vieles mehr. Instructions sind somit der Weg wie während der Kompilierung auf den QHScompiler einfluss genommen werden kann. In der Tabelle 4.2 sind ein paar der wichtigsten Instructions aufgelistet:

Tabelle 4.2: Wichtige Instructions des QHScompilers

#enterOrderQueue	Aktiviert die OrderQueue.
#exitOrderQueue	Deaktiviert die OrderQueue.
#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 Validate und somit die OrderQueue.
#orderEnqueue	Die nächste Order wird sofort der OrderQueue hinzugefügt, auch wenn diese orderQueue-proof wäre. Die Execute-Phase wird übersprungen.
#orderFrontEnqueue	Ähnlich wie #orderEnqueue. Die Order wird jedoch auf den obersten 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 oberste Order der OrderQueue wird oben auf den FetchStack gesetzt.
#pushEnv	Ein neues Environment wird der Environment Linked-List hinzugefügt.
#popEnv	Das neuste Environment der Environment Linked-List wird gelöscht.
#addTolIdentifier	Die oberste Order der OrderQueue muss ein Identifier sein und die zweitoberste LiteralCode. Beide Orders werden als Zahl interpretiert und addiert. Das Resultat wird im Identifier gespeichert.

Der QHScompiler umfasst **33** Instructions, wobei **5** dieser nur fürs Debugging des Compilers dienen.

4.3.3 LiteralCode

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 für den QHScompiler Assembly Code zu generieren. Somit könnte nur durch das Ändern einzelner LiteralCode Orders die Ausgabesprache des QHScompilers komplett geändert werden.

4.4 Definition der Macros

Somit ist der QHScompiler komplett. Grundsätzlich lässt sich mit QHS bereits jedes Programm schreiben. Jedoch habe ich in Tabelle 2.1 festgelegt, dass die Eingabesprache einen C ähnlichen Syntax aufweisen muss. Um dies zu ermöglichen, muss man, wie zu Beginn von Abschnitt 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 Abkürzungen

In diesem Abschnitt wird viel QHS-Code als Beispiel verwendet. Um die Leserlichkeit von QHS zu verbessern, werden ein zuerst paar Identifier anstelle der umständlichen Instructions definiert. Diese Identifier sind in der folgenden Tabelle 4.3 aufgeführt.

Tabelle 4.3: Identifier als Abkürzung von Instruction

[#enterOrderQueue
]	#exitOrderQueue
>>	#assign
->	#assignToOne
!	#force
\n	Eine neue Zeile im Ausgabedatei

Weiter wird innerhalb von Kommentaren 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 = "hello"#pushEnv */` würde bedeuten, dass der Identifier `X` zu den Orders `"hello"` (LiteralCode) und `#pushEnv` (Instruction) definiert wurde.

Auch wird bei längeren Identifier Definitionen wird zuerst der Identifier Name getrennt von den restlichen Orders der OrderQueue hinzugefügt. Diese Separation dient der besseren Leserlichkeit und hat keinen Einfluss auf die Kompilierung des Codes.

EXAMPLE

4.4.2 Identifier Parameter und Rückgabewert

Mit den `#enterOrderQueue` und `#exitOrderQueue` Instructions kann innerhalb eines Identifiers die OrderQueue verwendet werden. Dies ermöglicht eine Art von Parameter und Rückgabewert für Identifier. Parameter werden vor dem Aufruf eines Identifiers der OrderQueue hinzugefügt. Diese können dann innerhalb des Identifiers verwendet werden. Genauso kann der Identifier am Ende Orders der OrderQueue hinzufügen und diese somit zurückgeben.

Auflistung 4.1: Verwendung von Parametern und Rückgabewert eines Identifiers

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

  param1 " : " param2 \n          /* param1 + " : " + param2 + "\n" */
  [ "return" ]                    /* "return" wird der OrderQueue hinzugefügt */
] >>

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

Ausgabe

```
1 : 2
return
```

4.4.3 Variablen

Die Umsetzung von Variablen in QHS ist einfach. Zuerst muss die Grösse der Variable vom `rsp` 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 4.2: Definition einer Variable mit `LiteralCode`

```
"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 viel `LiteralCode` und Assembly Kenntnisse. Um die Definition von Variablen einfacher zu gestalten, 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 4.3: Definition einer Variable mit `var` Identifier

```
[ var ]
[
  #orderFrontEnqueue size ->      /* size = argument1 */
  [ name ! #deepFetch ] >>        /* name = Was nach dem var Identifier folgt */

  "sub rsp, " size \n

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

[ "4" ] var a

"add " a ", 5"
```

Ausgabe

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

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

Auflistung 4.4: Definition einer Variable mit rbpOffset

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

[ var ]
[
  #orderFrontEnqueue size ->                        /* size = argument1 */
  [ name ! #deepFetch ] >>                          /* name = Was nach dem var Identifier folgt */

  "sub rsp, " size \n

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

  [ ! name "[rbp-" ! rbpOffset "]" ] >>             /* name = "[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 4.5: Definition einer Variable mit *int* Identifier

```
(...)

[ 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 Setzen von Variablen werde ich hier nicht weiter eingehen, da dies mit den Methoden, die im nächsten Abschnitt 4.4.4 erklärt werden, funktioniert.

4.4.4 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 4.6: 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 4.4.3 wurde der *int* Identifier für die Definition einer Variable verwendet. Das *int* in der Auflistung 4.6 würde daher vom QHScompiler als Definition für eine Variable verstanden werden. Der Unterschied zwischen Variable 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 Variable 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 Abschnitt 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 Variable 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 4.7: Implementation eines DelayedExecute für Definitionen

(Definition einer Variable aus Auflistung 4.4)

```

[ function ]
[
  #orderFrontEnqueue returnSize ->          /* size = argument1 */
  #orderFrontEnqueue name ->                /* name = argument2 */

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

[ definition ]
[
  #orderFrontEnqueue size ->                 /* size = argument1 */
  [ name ! #deepFetch ] >>                  /* name = Was nach dem var Identifier folgt */

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

  [ ( ]
  [
    [ #orderEnqueue ! size #orderEnqueue ! 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 sollte bei einer Parameterdefinition nicht der LiteralCode zur Subtraktion vom `rsp` hinzugefügt werden. Zweitens verwendet eine Parameterdefinition einen anderen `rbp`-Offset. 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 4.8: Implementation eines TempAssigns für Parameter Definitionen

```
[ function ]
[
  #pushEnv

  #orderFrontEnqueue returnSize ->      /* size = argument1 */
  #orderFrontEnqueue name ->           /* name = argument2 */

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

  [ definition paramDefinition ]        /* definition = paramDefinition */

  #popEnv                               /* Umdefinition von definition wird vergessen */
] >>
```

Der Identifier *paramDefinition* ist ähnlich wie der *var* Identifier aus Abschnitt 4.4.3. Es wird bloß anstelle von *rbpOffset* ein neuer *paramOffset* Identifier verwendet und der Assembly Code fürs Subtrahieren vom *rsp* 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 4.9: Finale Definition einer Funktion in QHS

```
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 Abschnitt 2 beschrieben, mit zwei weiteren Compilern vergleichen. Verglichen 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 If-Else-Statements und Loops. Die Funktion wurde in die jeweiligen Sprachen übersetzt und mehrmals in das Programm eingefügt. Anschliessend wurde jedes Programm zehnmal kompiliert. Die durchschnittliche Dauer der Kompilierung ist in Abbildung 5.1 ersichtlich.

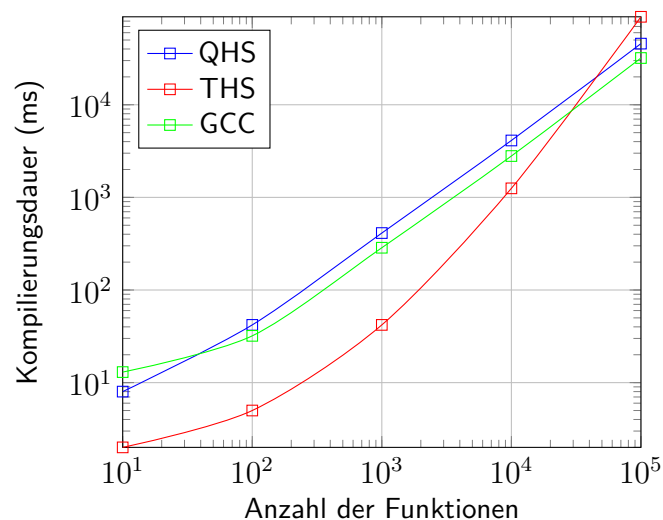


Abbildung 5.1: Vergleich der Kompilierungsdauer mit Log-Log Skalen

Interessant ist, dass sowohl der QHScompiler als auch GCC mit einer hohen Kompilierungsdauer bei einer tiefen Anzahl an Funktionen beginnen und sich später linear verhalten. Der THScompiler glänzt bei einer kleinen Programmgrösse mit einer sehr schnellen Kompilierung, doch steigt die Kompilierungsdauer mit einer steigenden Anzahl an Funktionen exponentiell an. Ab 10³ Kopien der Funktion wird GCC und daraufhin zwischen 10⁴ und 10⁵ Kopien auch der QHScompiler schneller als der THScompiler. Für die exponentielle Kompilierungsdauer des THScompilers habe ich leider keine Erklärung. Grundsätzlich sollten alle Schritte, die der THScompiler durchläuft, eine lineare Komplexität aufweisen. 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 10^2 Funktionskopien 7-8 mal länger als GCC. Der QHScompiler ist somit deutlich von GCC geschlagen. Wie GCC zeigt, liegt das Problem der exponentiellen Kompilierungsdauer des THScompilers nicht am traditionellen Compiler Aufbau und viel mehr an meiner Implementation davon. Daher überzeugt GCC und das traditionelle Compiler Model bei der Geschwindigkeit der Kompilierung mehr als mein alternativer Aufbau.

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 unterstützte 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.

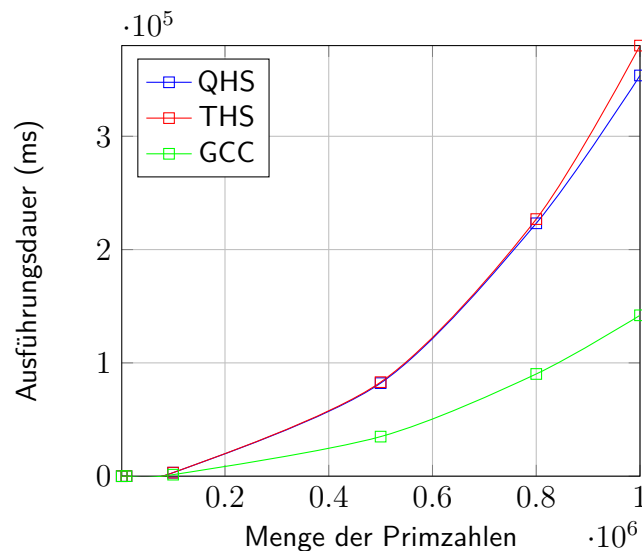


Abbildung 5.2: Vergleich der Ausführungsdauer mit GCC Optimierung

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:

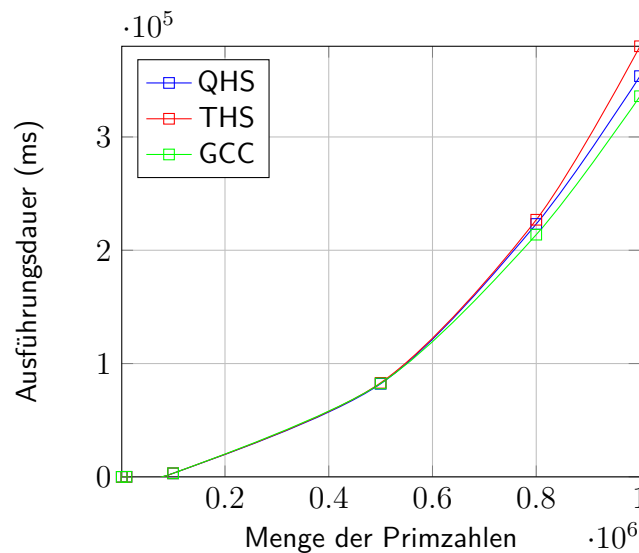


Abbildung 5.3: Vergleich der Ausführungsdauer ohne GCC Optimierung

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. Aus zeitlichen Gründen war es mir nicht möglich Optimierung in meine beiden Compiler einzubauen.

Zusammengefasst lässt sich sagen, dass ohne Optimierung mein alternativer Aufbau eines Compilers ungefähr gleich schnelle Ausführungsgeschwindigkeit liefert, wie ein traditioneller Compiler. Trotzdem muss man anmerken, dass Optimierung bei einem traditionellen Compiler möglich ist und die Ausführungsdauer deutlich verringert, wie GCC in Abbildung 5.2 beweist. Ich kann mir vorstellen, dass Optimierung für einen nach meinem alternativen Aufbau entwickelten Compiler jedoch deutlich schwierig wäre. Traditionelle Compiler haben die Möglichkeit Optimierungen auf dem AST auszuführen. Für meinem alternativen Compiler ist dies nicht möglich, da gar keine syntaktische Analyse durchgeführt und nie ein AST generiert wird. Auch steht die Grundidee meines alternativen Aufbaus Optimierung stark im Weg. Wie zu Beginn von Abschnitt 4 beschrieben, basiert mein Ansatz darauf, dass die von Macro Expansion verwendeten Macros während der Kompilierung erst 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 erscheinen dadurch manchmal etwas pingelig. Jedoch sind sie dafür sehr genau und hilfreich beim Finden und Melden von Fehlern.

Der QHScompiler arbeitet hingegen viel ungenauer. Wie im Abschnitt 4.4.4 bereits beschrieben, 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 führt bei QHScompiler auf eine Fehlermeldung. 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

```
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 OrderQueue-Empty Error 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 Compiler Aufbau mit syntaktischer und semantischer Analyse deutlich besser mit Fehlern umgehen kann als mein alternatives Model.

5.4 Offenheit für Erweiterung

C, als eine auch professionell verwendete Programmiersprache, umfasst eine Vielzahl an Features. Zum Beispiel lassen sich mittels Templates Datenstrukturen wie Stacks, Queues oder Vectors definieren, die Datentyp unabhängig sind. Mit Libraries lassen sich zudem komplexe Algorithmen einmal schreiben und später einfach wieder verwenden. Solche Funktionalitäten sind bei einem traditionellen Compiler möglich.

Mit dem QHScompiler lassen sich Libraries ebenfalls verwenden. Templates sollten theoretisch ebenfalls möglich sein, jedoch habe ich dies nicht getestet. Jedoch bietet der QHScompiler, wie in Abschnitt 4 bereits angemerkt, noch weitere Möglichkeiten zur Erweiterung. Das Definieren von Identifiern während der Kompilierung ermöglicht es ganz unterschiedliche Programmiersprachen mit dem QHScompiler zu kompilieren. Gegebenenfalls kann die Programmiersprache sogar innerhalb der Eingabedatei geändert werden. Leider ist die Definition der benötigten Identifier für eine Programmiersprache nicht besonders intuitiv und benötigt Methoden, wie die Abschnitt 4.4.4 erklärten `DelayedExecute` und `TempAssign`. Trotzdem würde ich sagen, dass der QHScompiler einem mehr Freiheit bei der Erweiterung bietet, als ein traditioneller Compiler.

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 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 sehr ineinander verschachtelt. (...)

Aus dem Vergleich der Umgang 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, folge der fehlenden syntaktischen Analyse, nicht möglich.

Ausserdem ist ein AST, wie in Abschnitt 5.2 thematisiert, auch für die Optimierung der Ausgabedatei äusserst praktisch. Grundsätzlich ist Optimierung für den QHScompiler äusserst schwierig. Da die Eingabesprache während der Kompilierung erst definiert wird, müssten auch passende Optimierungsmethoden spontan gefunden werden. Dies äussert sich in einer langsameren Geschwindigkeit der Ausgabedatei.

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 führt das System der während der Kompilierung definierten Identifier zu hohen Kompilierungsauern, ungenauem Umgang mit Fehlern und mangelhafter Optimierung der Ausgabedatei. Der QHScompiler ist einem traditionellen Compiler also stark unterlegen.

6 Schluss

Der QHScompiler kann einem traditionellen Compiler leider nicht das Wasser reichen. Trotzdem habe ich mit meiner Arbeit das erreicht, was ich mir erhofft hatte. Der Auslöser für meinen alternativen Ansatz war die Frage: Wieso werden Compiler so entwickelt, wie sie entwickelt werden? 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 ihr nicht auf verwirrende Methoden wie `DelayedExecute` oder `TempAssign` aus Abschnitt 4.4.4 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 mit guten Grund dem traditionellen Aufbau. Der QHScompiler ist somit kein bahnbrechender super Compiler, sondern eher ein Experiment. Ein Test für den traditionellen Aufbau von Compilern. Ein Test, den die traditionellen Compiler problemlos bestanden haben.

Es hat mir sehr viel Spass gemacht meinen alternativen Ansatz zu entwickeln. Auch wenn es häufig frustriert, 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 stiess ich häufig gegen eine Wand und bemerkte dies meist erst, als ich schon eine Woche daran 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 *Compiler Construction* der Universität Bern für diese Maturaarbeit war. Ohne diese wäre es für mich viel schwieriger gewesen den traditionellen Aufbau eines Compilers zu verstehen und umzusetzen. Auch basiert der Grossteil des Abschnitts 3 auf dieser Vorlesung. Wahrscheinlich wäre ich auch ohne die Vorlesung nie zu meinen alternativen Aufbau für Compiler inspiriert worden.

Abbildungsverzeichnis

3.1	Schritte, die ein Compiler durchläuft (https://github.com/munificent/craftinginterpreters , besucht am 5.8.2024)	6
3.2	Schritte, die in dieser Arbeit behandelt werden (Basierend auf Abbildung 3.1)	6
3.3	Abstract Syntax Tree (https://en.wikipedia.org/wiki/Abstract_syntax_tree , besucht am 5.8.2024)	8
4.1	Zyklus der QHS Kompilierung (QHS-Zyklus)	10
5.1	Vergleich der Kompilierungsdauer mit Log-Log Skalen	20
5.2	Vergleich der Ausführungsdauer mit GCC Optimierung	21
5.3	Vergleich der Ausführungsdauer ohne GCC Optimierung	22

Auflistungsverzeichnis

3.1	C code vor der lexikalischen Analyse	7
3.2	Tokens nach der lexikalische Analyse	7
4.1	Verwendung von Parametern und Rückgabewert eines Identifiers	14
4.2	Definition einer Variable mit LiteralCode	15
4.3	Definition einer Variable mit <i>var</i> Identifier	15
4.4	Definition einer Variable mit <i>rbpOffset</i>	16
4.5	Definition einer Variable mit <i>int</i> Identifier	16
4.6	Ziel für die Definition einer Funktion in QHS	17
4.7	Implementation eines DelayedExecute für Definitionen	18
4.8	Implementation eines TempAssigns für Parameter Definitionen	19
4.9	Finale Definition einer Funktion in QHS	19
5.1	QHS mit fehlenden Tokens	23
5.2	QHS mit fehlender öffnender Klammer	23
5.3	QHS mit falscher Anzahl Argumente	23

Tabellenverzeichnis

2.1	Anforderungen an die Compiler	5
2.2	Vergleichskriterien der Compiler	5
4.1	RegEx Definitionen der Ordertypen	11
4.2	Wichtige Instructions des QHScompilers	13
4.3	Identifizier als Abkürzung von Instruction	14

Literaturverzeichnis

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2006.
- [2] Susan L. Graham. *Table-Driven Code Generation*. 1980. [Online; accessed 2024-09-07].