Maturaarbeit

Compiler Construction

Fabio Stalder

Betreut durch
Thomas Jampen

24. September 2024

Gymnasium Kirchenfeld
Abteilung MN

Inhaltsverzeichnis

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 den resultierenden Output-Code. Dieser sollte optimal generiert werden, um die schlussendliche Ausführungsdauer so kurz wie möglich zu halten. Sollte sich doch einmal ein Fehler im Input-Code befindet, 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.

2 Vergleich der Compiler

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

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. Der traditionelle Compiler Aufbau wird in diesem Vergleich von GCC repräsentiert.

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.

2.1 Anforderungen an die Compiler

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

Tabelle 2.1: Anforderungen an die Compiler

Output als Assembly Code	Die Ausgabesprache muss Assembly Code sein
C-like 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

2.2 Kriterien des Compiler-Vergleichs

Zum Schluss werden die drei Compiler nach folgenden Kriterien bewertet und verglichen.

Tabelle 2.2: Vergleichskriterien der Compiler

Geschwindigkeit der Kompilierung	Wie lange dauert die Kompilierung von Code?
Geschwindigkeit des Output-Codes	Wie lange dauert die Ausführung des Output-Codes?
Benutzerfreundlichkeit	Wie einfach ist die Verwendung des Compilers?
Offenheit für Erweiterung	Wie einfach kann die Eingabesprache erweitert werden?

3 Ein traditioneller Compiler

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

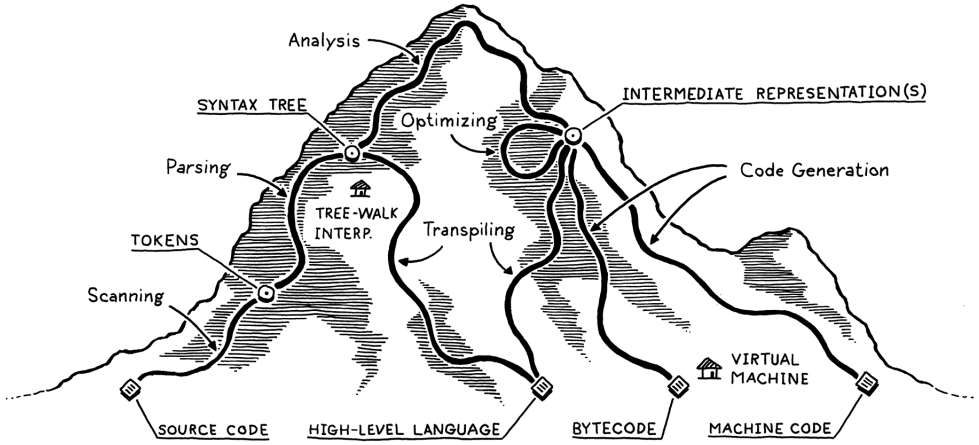

Abbildung 3.1: Schritte, die ein Compiler durchläuft

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

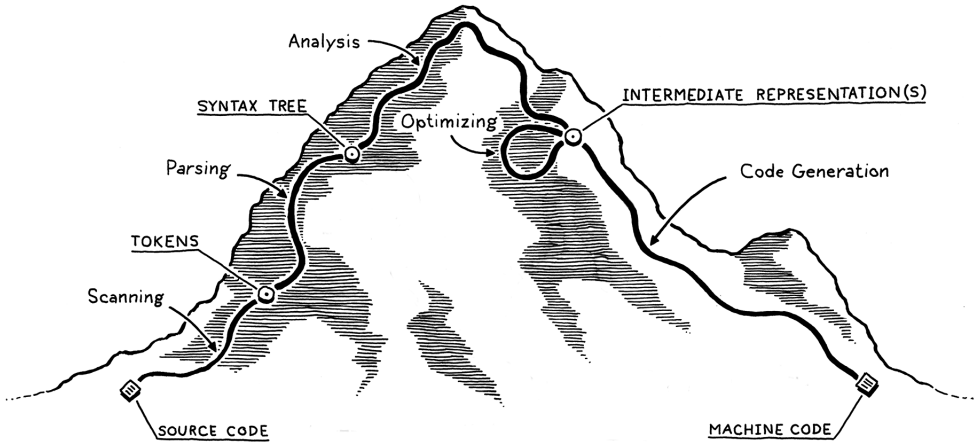

Abbildung 3.2: Schritte, die in dieser Arbeit behandelt werden

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

Als Basis für diese Kapitel dient grösstenteils eine Vorlesung der Universität Bern, die ich für freundlicherweise besuchen durfte.

3.1 Lexical Analysis

Meist werden Programme so geschrieben, dass wir Menschen sie lesen und verstehen können. Dafür verwendet man Buchstaben, Zahlen, Zeichen (wie + oder *) und Whitespaces (wie Leerzeichen oder Absätze). Diese sind jedoch für den Computer nicht sofort verständlich. Der erste Schritt beim Kompilieren ist daher die *Lexical Analysis*. Diese Analyse wird von einem Teil des Compilers, dem *Lexer*, durchgeführt. Die Aufgabe dieses Lexers ist es, die Inputdatei 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 Lexical Analysis auf der Programmierspache C:

Auflistung 3.1: C code vor Lexical Analysis

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

    return 1;
}
```

Auflistung 3.2: Tokens nach Lexical Analysis

```
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 Input-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 KeywordToken gesehen wird, lässt sich im Lexer auch bestimmen, dass ein Wort wie 'else' als Keyword angesehen werden soll.

3.2 Syntax Analysis

Nun versteht der Compiler, was mit den Zeichen in der Inputdatei gemeint ist. Es fehlt jedoch noch das Verständnis für den Syntax der Input-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 *Syntax Analysis* durch. Dabei werden die bei der Lexical Analysis gefundenen Tokens ineinander verschachtelt und in einen sogenannten *Abstract Syntax Tree (AST)* überführt.

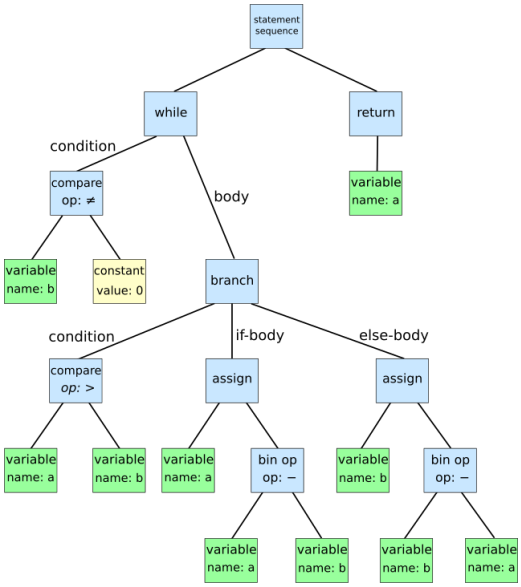

Abbildung 3.3: Abstract Syntax Tree zum Euklidischen Algorithmus

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 Semantic Analysis

Semantik ist die Wissenschaft der Bedeutung von Wörtern einer Menschensprache. Bei einem Compiler geht es bei der *Semantic Analysis* 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 Semantic Analysis entdeckt und gemeldet. Auch werden unbekannte Variablen und Funktionen in diesem Schritt abgefangen. Weiter wird der Datentyp einer Node an diese angebunden. Gegebenenfalls kann auch 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.

3.4 Code Generation

Code Generation ist der finale und oft auch komplexeste Schritt, der ein Compiler ausführen muss. Nun da unser Input-Code nicht mehr nur als Textfile, sondern als Intermediate Representation vorliegt, kann endlich Output-Code generiert werden. Eine geläufige Methode der Code Generation ist die sogennante

Macro Expansion. Hierbei wird der AST von unten nach oben schrittweise mit Teilen an Output-Code ersetzt. Diese Output-Code Teile sind häufig von den darunterliegenden Nodes abhängig.

MAYBE FIGURE OR LISTING AS EXAMPLE

3.5 Optimization

Code Generation ist zwar der letzte Schritt beim Kompilieren, trotzdem wurde eine wichtige Aufgabe des Compilers noch nicht betrachtet. *Optimization* geschieht zwischen jedem der genannten Schritte und dies häufig mehrmals. Dabei geht es darum den Output-Code so effizient wie möglich zu machen. Effizient kann hierbei Verschiedenes bedeuten. Der Output-Code muss so schnell wie möglich ausgeführt werden, Memory sparsam verwenden und dazu noch eine möglichst kleine Datei sein. Optimization reicht vom Entfernen 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. Compiler Optimization ist also sehr vielseitig und komplex. Wie Optimization 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 für Macro Expansion aus Abschnitt 3.4 sollen während der Kompilierung erst definiert werden. **Aus dieser Grundidee lassen zwei Dinge aufbauen.** Erstens halte ich es für möglich, mit der richtigen Verwendung von Macros die gesammte Syntax Analysis 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 definieren.

Die Sprache, in der sich Macros definieren lassen, wird als QHS bezeichnet. Diese Programmiersprache besteht wie die meisten anderen 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.

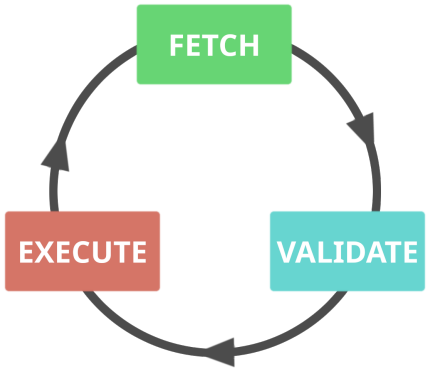

Abbildung 4.1: Zyklus der QHS Kompilierung (QHS-Zyklus)

4.1 Fetch

Der *QHS-Zyklus* beginnt mit *Fetch*. Die Aufgabe von *Fetch* ist es die nächste Order, die verarbeitet werden soll, zu finden. In dieser Hinsicht gleicht Fetch der Lexical Analysis eines traditionellen Compilers. Beim ersten Fetch wird die erste Order aus der Inputdatei extrahiert. Bei jedem weiteren Fetch wird nun die nächste Order aus der Inputdatei geholt. Eine Order weist wie erwähnt einen der drei Typen Identifier, Instruction oder LiteralCode auf. Fetch ist dafür zuständig die nächste Order einem der drei Typen zuzu-

ordenen. Diese Ordertypen sind mit folgenden RegEx definiert. Whitespaces dienen als Trennung zwischen zwei Orders und werden ignoriert.

Tabelle 4.1: RegEx Definitionen der Ordertypen

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

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

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

Bei einem Fetch kommt die nächste Order wie erwähnt von der Inputdatei. Es ist jedoch möglich Orders voranzustellen. Diese Orders werden beim nächsten Fetch zuerst gefunden. Dies geschieht mithilfe des *FetchStacks*, auf den Orders gelegt werden können. Beim nächsten Fetch wird immer die oberste Order des FetchStacks geholt und daraufhin vom FetchStack enfernt. Die Inputdatei befindet sich auf dem untersten Platz des FetchStacks und wird somit nur verwendet, wenn der Stack ansonsten komplett leer ist. Eine Order kann während jedem der drei Schritte des Zyklus auf den FetchStack gelegt werden. 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 Validate

Nachdem die nächste Order durch Fetch gefunden wurde, wird diese Order an Validate weitergegeben. Während dem Validate Schritt kommt die *OrderQueue* ins Spiel. Hierbei 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 den Validate Schritt gelangt und die OrderQueue aktiviert ist, wird diese Order der OrderQueue hinzugefügt. Der Execute Schritt wird danach übersprungen und der QHS-Zyklus beginnt von neuem bei Fetch. Die Order wurde, ohne Execute 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 Execute weitergegeben, auch wenn die OrderQueue aktiv ist. Dieses Prinzip ist zum Beispiel besonders bei der Instruction, die die OrderQueue wieder deaktiviert, wichtig. Da diese Instruction ansonsten nicht zu Execute 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 Execute

Execute ist der letzte Schritt des QHS-Zyklus. Hier wird der tatsächliche Assembly Code generiert. Je nach Typ der Order (Identifier, Instruction oder LiteralCode) läuft Execute sehr unterschiedlich ab.

4.3.1 Identifier

Ein Identifier ist eine Referenz zu den an Anfang von Abschnitt 4 erwähnten Macros. Diese sind in einem *Environment* definiert. Hierbei handelt es sich um eine einfache Map, die einen Identifier mit einer Liste an Orders verknüpft. Wenn nun ein Identifier in den Execute Schritt kommt, werden die dazugehörigen Orders auf den FetchStack aus Abschnitt 4.1 gelegt. Bei den nächsten Fetches werden nun zuerst die zum Identifier gehörenden Orders nacheinander abgebaut. Einfach ausgedeückt wird der Identifier im Inputfile mit seinen Orders ersetzt.

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

Instructions sind die komplexesten Orders für den Execute Schritt. Für jede Instruction ist im QHScompiler eine Funktion definiert, die ausgeführt wird, wenn diese Instruction in den Execute Schritt gelangt. 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 Fetch sofort an Execute weitergegeben. Überspringt Validate und somit die OrderQueue.
#lightForce	Ähnlich wird #force, jedoch wird diese nur ausgeführt, wenn **explain this cuz they don't know OrderQueue depth**
#orderEnqueue	Die nächste Order wird sofort der OrderQueue hinzugefügt, auch wenn diese Order orderQueue-proof wäre. Execute wird übersprungen.
#orderFrontEnqueue	Ähnlich wie #orderEnqueue. Die Order wird jedoch auf den obersten Platz der OrderQueue gesetzt.
#deepFetch	Die nächste Order der Inputdatei wird oben auf den FetchStack gesetzt. Ermöglicht den Zugriff auf die Inputdatei 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.
#addLiterals	Die beiden obersten Orders auf der OrderQueue müssen LiteralCode sein. Diese werden als Zahl interpretiert und addiert. Sollten diese keine Zahl sein, wird ein Fehler gemeldet.

Der QHScompiler umfässt **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 den Execute Schritt gelangt, wird alles was zwischen den Satzzeichen steht in das Output-Dokument geschrieben. Dies ist die einzige Möglichkeit für den QHScompiler Assembly Code zu generieren. Somit könnte nur durch das Ändern einzelner LiteralCode Orders die Output-Sprache des QHScompilers komplett geändert werden.

4.4 Bringing it all together

Und somit ist der QHScompiler komplett. Im Vergleich zu einem traditionellen Compiler wirkt der QHScompiler fast schon zu simpel. (... cuz function and vars and all of that hasn't even been mentioned) Dies hat einen einfachen Grund. Der QHScompiler ist zwar komplett, die dazugehörige Programmiersprache QHS jedoch noch lange nicht. Grundsätzlich ist es möglich mit LiteralCode jedes Programm zu schreiben und zu kompilieren, jedoch handelt es sich dann nur um Assembly Code. Doch der Aufbau des QHScompilers ermöglicht es mit Identifiern eine komplexere Programmiersprache zu definieren. Ein fester Bestandteil ein jedes Programms, das mit dem QHScompiler kompiliert werden soll, ist ein Stück

Code, das die jeweilige Programmiersprache definiert. Dieser Code wird im Kontext des QHScompilers *Preamble* genannt. Theoretisch ist es möglich durch das Anpassen dieses Preambles, viele unterschiedliche Programmiersprachen mit dem QHScompiler zu kompilieren. In diesem Abschnitt wird behandelt wie sich die Sprache QHS, welche die Kriterien aus Abschnitt 2.2 erfüllt, für den QHScompiler definieren lässt.

4.4.1 Abkürzungen

Um die Leserlichkeit von QHS zu verbessern, werden ein paar Identifiers anstelle der umständlichen Instructions definiert. Diese sind in der folgenden Tabelle 4.3 aufgeführt.

Tabelle 4.3: Identifiers als Abkürzung von Instruction

`[`	#enterOrderQueue
`]`	#exitOrderQueue
`>>`	#assign
`->`	#assignToOne
`!`	#force
`?!`	#lightForce
`\n`	Eine neue Zeile im Outputdatei

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.

Besonders 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.

4.4.2 Identifier Parameter und Rückgabewert

Mithilfe der #enterOrderQueue und #exitOrderQueue Instructions kann innerhalb eines Identifiers die OrderQueue verwendet werden. Dies ermöglicht eine Art von Parametern und Rückgabewert für Identifier. Parameter werden vor dem Aufruf eines Identifiers der OrderQueue hinzugefügt. Diese kann dann der Identifier verwenden. 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 */
```

Output

```
1 : 2
return
```

4.4.3 Variablen

Die Umsetzung von Variablen in QHS ist einfach. Zuerst soll der Assembly Code für das Abziehen der Grösse der Variable vom Stack-Pointer hinzugefügt 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"
```

Output

```
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 gestallten, 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 vielen Programmiersprachen 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]" ] >>            /* var = "[rbp-4]" */
] >>

[ "4" ] var a

"add " a ", 5"
```

Output

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

Momentan erhält jede Variable jedoch noch die Addresse 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 #addToIdentifier 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 "]" ] >>     /* var = "[rbp-OFFSET]" */
] >>

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

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

Output

```
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"
```

Output

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

Nun sieht die Definition einer Variable genau so aus, wie es in anderen Programmiersprachen gebräuchlich ist.

4.4.4 Funktionen

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 erstes 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 vom QHScompiler also 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. Dieses Vorgehen ist jedoch aufgrund des einfachen Designs des QHScompilers nicht möglich. Er kann bloss Orders ausführen, nicht jedoch überprüfen, ob eine Order vorhanden ist. 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. Weiter wird eine Klammer als Identifier für eine Funktionsdefinition gesetzt, sowie 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

```
[ 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 eine 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 gleich wie der *var* Identifier aus Abschnitt 4.4.3. Jedoch wird anstelle von *rbpOffset* ein neuer *paramOffset* Identifier verwendet.

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 Outputdatei 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
}

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

Mit von DelayedExecute und TempAssign lässt sich also syntaktisch komplexen Code problemlos in QHS definieren und ausführen.

QHS weist noch viele weitere Identifier, die einen C-like Syntax ermöglichen, auf. Diese Identifier werden hier jedoch nicht weiter betrachtet, da diese einem ähnlichen Prinzip wie die beschriebenen Definitionen von Variablen und Funktionen folgen.

5 Auswertung

Abschliessend will ich den QHScompiler, wie in Abschnitt 2 beschrieben, mit zwei weiteren Compilern vergleichen. Diese werden in Geschwindigkeit der Kompilierung, Geschwindigkeit des Output-Codes, Benutzerfreundlichkeit und Offenheit für Erweiterung bewertet.

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 5.1 ersichtlich.

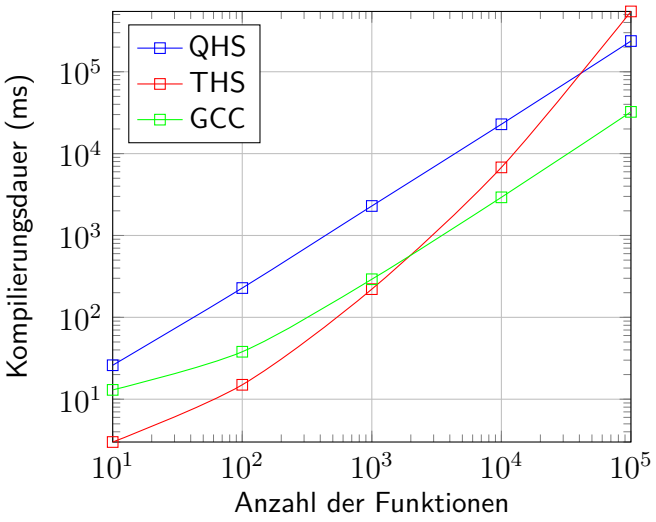

Abbildung 5.1: Vergleich der Kompilierungsdauer mit Log-Log Skalen

Interessant ist hierbei, dass sowohl QHS als auch GCC mit einer hohen Kompilierungsdauer beginnen und sich später linear verhalten. Während THS zwar zu Beginn mit einer sehr schnellen Kompilierung glänzt, daraufhin jedoch exponentiell ansteigt. Bei etwas mehr als 10^3 Kopien der Funktion wird GCC und daraufhin zwischen 10 und 10 Kopien auch der QHScompiler schneller als der THScompiler. Für die exponentielle Kompilerungsdauer des THScompilers habe ich leider keine Erklärung. Grundsätzlich sollten alle Schritte, die der THScompiler durchläuft, eine lineare Komplexität aufweisen. Daher liegt der Fehler wahrscheinlich bei meinen eigenen C++ Kenntnissen. Durch die logarithmischen Skalen erscheint der Unterschied zwischen den Kompilierungsdauern von GCC und dem QHScompiler konstant, jedoch braucht

der QHScompiler ab einer Programmgrösse über 10^2 Funktionskopien konsistent 7-8 mal länger als GCC. Der QHScompiler ist somit deutlich geschlagen. Wie GCC zeigt, liegt das Problem der exponentiellen Kompilierungsdauer beim THScompiler nicht am Prinzip des traditionellen Compilers und viel mehr an meiner Implementation davon. Daher würde ich in dieser Kategorie des Vergleichs den Sieg für den traditionellen Compiler aussprechen.

5.2 Geschwindigkeit des Output-Codes

Die Geschwindigkeit eines kompilierten Programmes wird anhand eines Algorithmus zur Berechnung von Primzahlen gemessen. Sowie bei der Funktion aus Abschnitt 5.1 ist dieser Algorithmus so geschrieben, dass er möglichst jedes von allen drei Compilern unterstütze Feature verwendet. Dieser Algorithmus wurde von Hand in die jeweiligen Sprachen übersetzt.

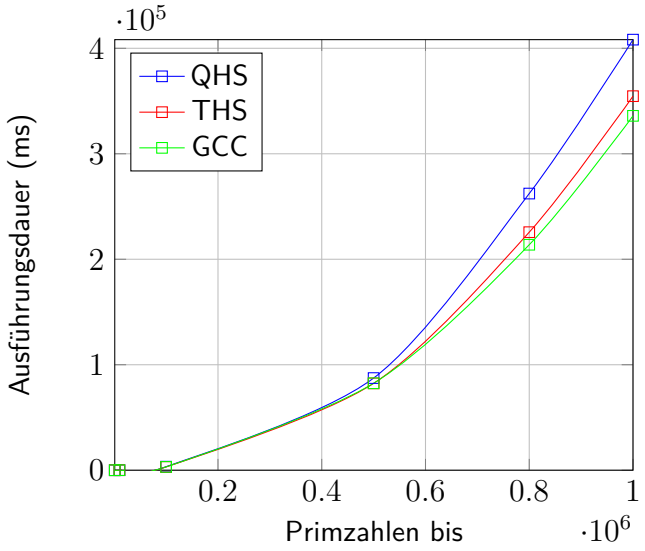

Abbildung 5.2: Vergleich der Ausführungsdauer

In Grafik 5.2 ist ersichtlich, dass (...)

5.3 Benutzerfreundlichkeit

Benutzerfreundlichkeit ist im Gegensatz zu den beiden vorherigen Vergleichskriterien etwas Subjektives. Jedoch würde ich behaupten, dass auch hier das Urteil ziemlich klar ist. GCC und der THScompiler folgen beide exakt definiertem Syntax und Semantik. Dies ist ein Resultat des Lexers und des Parsers die noch diesen bestimmten Regeln geschrieben wurden. Anfangs scheinen Semikolons am Ende jedes Statements vielleicht etwas unnötig, jedoch bemerkt man schnell, dass genau diese Pingeligkeit der Compiler für eine Programmiersprache äusserst wichtig ist. GCC fängt besonders gut Fehler früh ab und meldet diese. Der traditionelle Compiler ist somit sehr gut in puncto Benutzerfreundlichkeit.

Der QHScompiler weisst hier hingegen einige Macken auf. 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 ganz einfach und strickt nur aus was als Nächstes auftaucht. Somit führt ein fehlendes Zeichen nicht immer zu Fehlern. Folgendes Beispiel kompiliert einwandfrei und lässt sich auch problemlos ausführen.

Auflistung 5.1: QHS mit fehlenden Tokens

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

Weder das Semikolon noch die schliessende Klammer bei 5.1 ist hierbei nötig und das Programm lässt sich problemlos kompilieren und ausführen. Jedoch kann dies auch anders laufen.

Auflistung 5.2: QHS mit fehlender (

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

Der QHS Code bei 5.2 kompiliert einwandfrei, jedoch ist der genierte Assembly Code fehlerhaft. Die Funktion foo wird nicht ausgeführt und die Variable a nicht als Argument angesehen.

Weiter sind auch die Fehlermeldungen des QHScompilers nicht immer besonders klar.

Auflistung 5.3: QHS mit falscher Anzahl Argumente

```
void foo ( ) { }

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

    exit ;
}
```

Output

```
[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 5.3 wird die Funktion foo ohne Parameter definiert, später jedoch mit einem Argument aufgerufen. Der QHScompiler verfügt hierbei über keine Möglichkeit die Menge an Argumenten zu überprüfen und meldet nicht direkt einen Fehler. Als er jedoch versucht die Grösse des erwarteten Argumentes von der OrderQueue zu nehmen ist diese leer. Der QHScompiler meldet also einen OrderQueue-Empty Error gefolgt von vielen Folgefehlern.

Somit ist der QHScompiler bei der Meldung von Fehlern einerseits weniger strikt, andererseits aber auch deutlich verwirrender und ungenauer als ein traditioneller Compiler. In meinen Augen triumphiert daher auch in dieser Kategorie der traditionelle Compiler über meinen QHScompiler.

5.4 Offenheit für Erweiterung

Als eine auch professionell verwendete Programmiersprache, hat C selbstverständlich eine Vielzahl an Features. Zum Beispiel lassen sich mithilfe von Templates Datentyp unabhängige Datenstrukturen wie Stacks, Queues oder Vectors definieren. Weiter lassen sich mit Libraries komplexe Algorithmen einmal schreiben und nachher ganz einfach wieder verwenden. All dies ist innerhalb eines traditionellen Compilers möglich.

Libraries lassen sich ebenfalls mit dem QHScompiler verwenden. Templates sollten theoretisch ebenfalls möglich sein, jedoch habe ich dies nicht getestet. Jedoch unterstützt der QHScompiler, wie in Abschnitt 4 bereits angetönt, noch weitere Möglichkeiten zur Erweiterung. Denn es ist möglich eigene Identifier zu definieren. Mit den im Abschnitt 4.4.4 beschriebenen Techniken DelayedExecute und TempAssign lassen sich sogar selbstständig syntaktisch komplexe Code Strukturen bilden. Im Gegensatz zu einem traditionellen Compiler muss hierfür nicht einmal der QHScompiler angepasst werden. Dadurch kann man unterschiedliche Programmiersprachen mit dem QHScompiler kompilieren. Es ist sogar möglich die Programmiersprache innerhalb einer Datei zu wechseln. Leider ist die Definition der benötigten Macros für eine neue Programmiersprache nicht besonders intuitiv. 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 hat der QHScompiler nicht besonders gut abgeschnitten. Er ist sowohl in der Geschwindigkeit der Kompilierung als auch bei der eines kompilierten Programmes einem traditionellen Compiler unterlegen. Zudem ist der QHScompiler auch nicht besonders benutzerfreundlich und nicht wirklich angenehm zu verwenden. Als einziger Vorteil lässt sich seine Möglichkeit zur 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 Identifiers häufig so verschachtelt, dass

Aus dem Vergleich der Benutzerfreundlichkeit wird ausserdem klar, dass die Syntax Analysis 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 Syntax Analysis 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 des Output-Codes.

Der einzige Vorteil des QHScompilers liegt in der Offenheit für Erweiterung. Das Wechseln der Programmiersprache innerhalb einer Datei ist definitv 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 Kompilerung definierten Identifier zu hohen Kompierungsdauern, ungenauem Umgang mit Fehlern und mangelhafter Optimierung des Output-Codes. Der QHScompiler ist einem tradtionellen Compiler also stark unterlegen.

6 Schluss

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] Susan L. Graham. *Table-Driven Code Generation*. 1980. [Online; accessed 2024-09-07].