

**Maturaarbeit**

# **Compiler Construction**

Fabio Stalder

Betreut durch  
Thomas Jampen

28. August 2024



Gymnasium Kirchenfeld  
Abteilung MN

# Inhaltsverzeichnis

<b>1</b>	<b>Was ist ein Compiler</b>	<b>3</b>
1.1	Lexical Analysis . . . . .	3
1.2	Syntax Analysis . . . . .	4
1.3	Semantic Analysis . . . . .	5
1.4	Code Generation . . . . .	5
1.5	Optimization . . . . .	5
<b>2</b>	<b>Meine Idee</b>	<b>6</b>
<b>3</b>	<b>Vergleich der Compiler</b>	<b>7</b>
3.1	Anforderungen an die Compiler . . . . .	7
3.2	Kriterien des Vergleichs . . . . .	7
<b>4</b>	<b>THS Compiler</b>	<b>8</b>
<b>5</b>	<b>QHS Compiler</b>	<b>9</b>
5.1	Fetch . . . . .	9
5.2	Decode . . . . .	10
5.3	Execute . . . . .	11
5.3.1	Identifizier . . . . .	11
5.3.2	Literal-Code . . . . .	11
5.3.3	Instructions . . . . .	11
5.4	Bringing it all together . . . . .	12
5.4.1	Shortcuts . . . . .	12
5.4.2	Variablen . . . . .	12

# 1 Was ist ein Compiler

In der Informatik beschreibt Compiler ein Programm, das Code aus einer Programmiersprache in eine andere übersetzt. In dieser Hinsicht gleichen Compiler Übersetzern für Menschensprache. Jedoch unterscheidet sich ein Compiler grundsätzlich von Übersetzern in der Erwartungshaltung, die an sie gestellt wird. Menschensprache ist sehr komplex und [...]

Ein Compiler ist traditionell nach folgendem Schema aufgebaut.

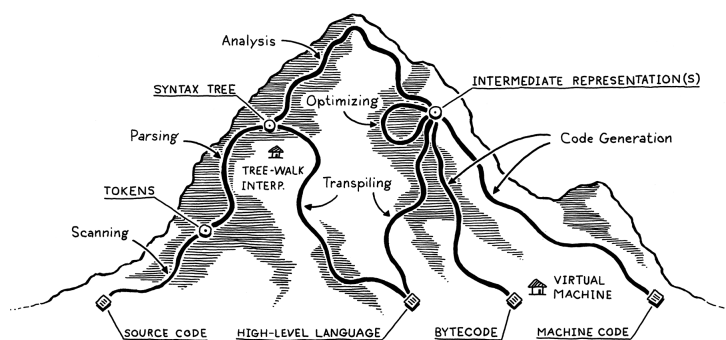


Abbildung 1.1: Schritte, die ein Compiler durchläuft [1]

In dieser Arbeit werde ich mich nur auf die im unteren Schema dargestellten Schritte fokussieren.

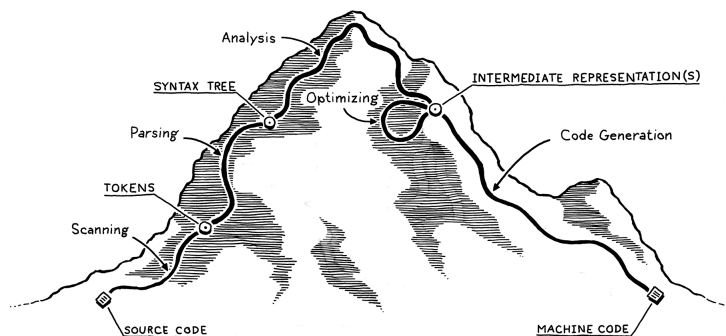


Abbildung 1.2: Schritte, die in dieser Arbeit behandelt werden (Basierend auf Figure 1.1)

## 1.1 Lexical Analysis

Meist werden Programme so geschrieben, dass wir Menschen es lesen und verstehen können. Dafür verwendet man Buchstaben und Zahlen, Zeichen, wie +, \*, oder Klammern, und Whitespaces, wie Leerzeichen oder Absätze. Diese Zeichen sind jedoch für den Computer unverständlich. Der erste Schritt beim compilieren ist daher die Lexical Analysis. Dies wird von einem Teil des Compilers, dem Lexer, durchgeführt.

Die Aufgabe dieses Lexers ist es den Input File zu scannen und die gescannten Zeichen in sogenannte Tokens zu verwandeln. Diese Tokens sind Datenstrukturen, die der Compiler kennt und mit denen er weiterarbeiten kann.

Als Beispiel:

Listing 1.1: C code vor Lexical Analysis

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

    return 1;
}
```

Würde hierbei zu einem Array von Token Objekten umgewandelt werden:

Listing 1.2: Tokens nach Lexical Analysis

```
KeywordToken (keyword="int")
IdentifierToken (id="foo")
LParenthesisToken
RParenthesisToken
KeywordToken (keyword="if")
LParenthesisToken
IdentifierToken (id="bar")
OperatorToken (operator=ComparisonEqual)
LiteralIntToken (value=0)
[...]
```

Der Lexer legt hierbei 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 1.2 'if' als KeywordToken gesehen wird, lässt sich im Lexer auch bestimmen, dass ein Wort wie 'print' als Keyword angesehen werden soll.

## 1.2 Syntax Analysis

Nun versteht der Compiler was mit den Zeichen im Input File gemeint ist, jedoch fehlt noch etwas bis tatsächlich in eine andere Programmiersprache übersetzt werden kann. Und das ist Verständnis für Syntax. Die meisten High-Level Programmiersprachen weisen Syntaxregeln auf. Diese beinhalten, wie Funktionen und Variablen definiert werden oder mit welchen Punktvorstrich-Regeln Expressions evaluiert werden. Die bei der Lexical Analysis gefundenen Tokens werden nun ineinander verschachtelt und in einen sogenannten Abstract Syntax Tree (AST) überführt.

Ein AST enthält somit nicht nur Informationen über die Tokens, sondern über die gesamte Struktur die sich aus den Tokens ergibt. Variabel- und Funktionsdefinitionen oder komplexe Statements wie 'if' oder 'for' sind hierbei im AST enthalten. [...]

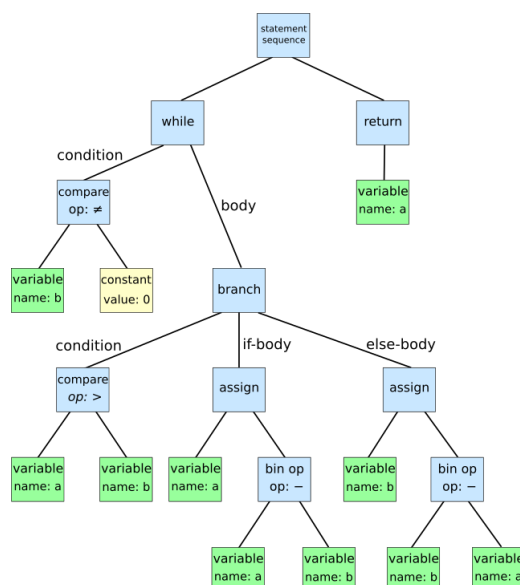


Abbildung 1.3: Abstract Syntax Tree zum Euklidischen Algorithmus [2]

### 1.3 Semantic Analysis

Wie auch bei den meisten Menschensprachen gibt es auch für Programmiersprachen eine Semantik und diese muss natürlich vom Compiler verstanden werden. [...]

### 1.4 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. Jedoch lässt sich über diesen Schritt fast am wenigsten sagen, da er je nach Output-Sprache sehr unterschiedlich aussehen kann.

### 1.5 Optimization

Code Generation ist zwar der letzte Schritt beim Compilieren, trotzdem wurde eine wichtige Aufgabe des Compilers noch nicht betrachtet. Optimization ist eine Sprache die zwischen jedem der genannten Schritte geschieht. Dabei geht es darum den Output-Code so effizient wie möglich zu machen. Effizient kann hierbei jedoch viel Verschiedenes bedeuten. Der Output-Code muss so schnell wie möglich ausgeführt werden können, Memory sparsam verwenden und am besten auch noch ein kleiner File sein. Optimization reicht vom Entfernen der Kommentare beim Scannen oder umstellen von mathematischen Operationen bis zu entfernen von ungebrauchten Variablen und Deadstores. Es muss von CPU Registern profitiert, mit Heap-Memory umgegangen und von inline Funktionen Gebrauch gemacht werden. Compiler Optimization ist somit ein sehr vielseitiges Problem, dass hierbei nicht weiter thematisiert werden sollte.

## 2 Meine Idee

**Wie im vorherigen Abschnitt gezeigt**, ist ein Compiler ein äusserts komplexes Programm, mit vielen verschiedenen Schritten. Jedoch ist die zugrundeliegende Aufgabe gar nicht so kompliziert. Man braucht ja nur, ein Dokument mit Text der bestimmten Regeln folgt, in Text mit anderen Regeln verwandeln. Natürlich ist dies etwas salopp ausgedrückt, trotzdem fragte ich mich, ob es nicht möglich sei einen viel einfacheren Compiler zu schreiben.

# 3 Vergleich der Compiler

Compiler sollen verglichen werden

## 3.1 Anforderungen an die Compiler

Um einen **fairen** Vergleich zu ermöglichen, müssen die Compiler folgende Anforderungen erfüllen.

Output als Assembly Code	Die Output-Sprache muss Assembly Code sein
C-like Syntax	Die Input-Sprache muss einen C-like Syntax aufweisen
Variablen und Funktionen	Lokale und globale Variablen sowie Funktionen müssen unterstützt werden
Benutzerdefinierte Datatypes	Benutzerdefinierte Datatypes müssen unterstützt werden

Die Anforderungen machen dass Compiler gleich komplex.

## 3.2 Kriterien des Vergleichs

Die Compiler werden nach folgenden Kriterien bewertet und verglichen.

Geschwindigkeit des Output-Codes	Wie schnell wird der Output-Code ausgeführt?
Geschwindigkeit der Compilation	Wie lange dauert die Compilation von Code?
Benutzerfreundlichkeit	Wie einfach ist die Verwendung des Compilers
Möglichkeit für Erweiterung	Wie einfach ist den Compiler oder die Input-Sprache zu erweitern?

## 4 THS Compiler

Der THS Compiler folgt dem theoretischen Aufbau eines Compilers und besteht aus Lexer, Parser und Code Generator. Als Parser wird ein Predictive Descent Parser verwendet. Der Code Generator arbeitet auf dem Abstract Syntax Tree mithilfe eines Visitor Patterns. Die Semantic Analysis wird während der Code Generation durchgeführt. Geschrieben ist der Compiler in C++ und liefert x86 Assembly nach NASM Syntax.



## 5 QHS Compiler

Genauso wie der THS Compiler ist auch der QHS Compiler in C++ geschrieben und generiert x86 Assembly nach NASM Syntax. Jedoch unterscheiden sich beide Compiler stark in der Funktionsweise. Der Compilation von QHS steht ein einfacher Zyklus zugrunde, dessen Vorbild der Von-Neumann Zyklus ist.

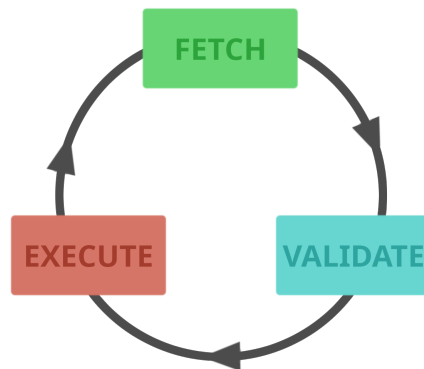


Abbildung 5.1: Zyklus der QHS Compilation

TALK ABOUT ORDERS AND THE MEANING OF IDENTIFIERS, INSTRUCTIONS AND LITERAL CODE

### 5.1 Fetch

Der QHS-Zyklus beginnt mit dem ersten Fetch. Dabei wird die erste Order aus dem Inputfile extrahiert. Eine Order weist einen der drei Typen Identifier, Instruction oder Literal-Code auf. Diese sind mit folgenden RegEx definiert.

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

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

Es ist hierbei möglich bestimmte Orders **voraus zu stellen**, die anstelle der nächsten Order im Inputfile gefetched werden. Dies geschieht mit Hilfe des Fetch-Stacks auf den eine Liste an Orders gepushed werden kann. Dieser Fetch-Stack folgt Last-In First-Out und **auf ihn** kann während jeder der drei Schritte des Zyklus, meist jedoch während Execute, gepushed werden. Die Hauptanwendung des Fetch-Stacks wird im Abschnitt 5.3 ausgeführt.

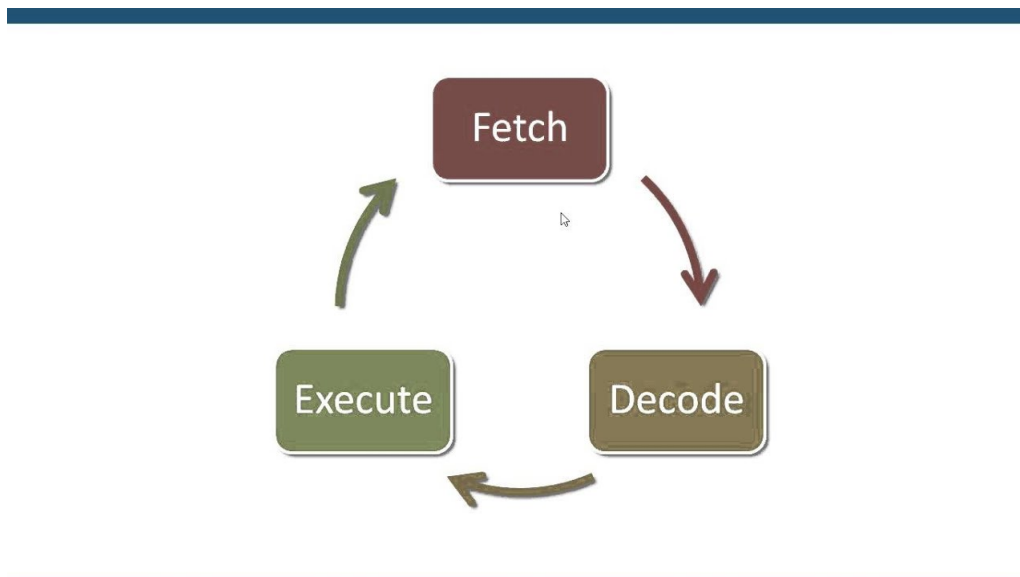


Abbildung 5.2: TEMP! Struktur des Fetch-Stacks

Wenn eine Liste an Orders komplett gefetched wurde, wird diese vom Stack gelöscht. Der Inputfile befindet sich auf dem letzten Platz des Fetch-Stacks und wird somit nur verwendet, wenn der Stack ansonsten komplett leer ist. Die Compilation wird beendet, sobald keine Order mehr auf dem Fetch-Stack übrig ist.

## 5.2 Decode

Nachdem eine Order gefetched wurde, wird diese an Decode weitergegeben. **Während Decode werden zwei Aufgaben durchgeführt.**

Während dem Decode Schritt kommt die Order-Queue ins Spiel. Hierbei handelt es sich um eine Liste an Orders, der Form First-In First-Out. Die Anwendung Aufgabe der Order-Queue ist das Speichern und spätere Ausführen von Orders. Die Order-Queue kann mit Hilfe von Instructions, die im Abschnitt 5.3 weiter ausgeführt werden, aktiviert und deaktiviert werden. Wenn nun eine Order in den Decode Schritt gelangt und die Order-Queue aktiviert ist, wird diese Order der Order-Queue hinzugefügt. Der Execute Schritt wird danach übersprungen und der Zyklus beginnt von neuem bei Fetch. Die Order wurde ohne ausgeführt zu werden auf der Order-Queue gespeichert. Später ist es nun möglich diese Order mit Hilfe von Instructions, die im Abschnitt 5.3 weiter thematisiert werden, von der Order-Queue zu entfernen und auszuführen. Bestimmte Instructions und Identifiers können jedoch Order-Queue-Proof, also immun gegen die Order-Queue, gemacht werden. Diese werden, auch wenn die Order-Queue aktiv ist, normal an Execute weitergegeben. Dies ist zum Beispiel besonders bei der Instruction, die die Order-Queue wieder deaktiviert, wichtig. Da diese Instruction sonst nicht ausgeführt und somit die Order-Queue nie mehr deaktiviert wird. LiteralCode kann nicht Code-Queue-Proof sein.

MAYBE CODESTACK FIGURE

Ist die Order-Queue deaktiviert oder die Order Code-Queue-Proof wird diese an den letzten Schritt Execute weitergegeben.

## 5.3 Execute

Execute ist der letzte Schritt des Zyklus. Und hier wird nun auch endlich der tatsächliche Assembly Code generiert. Je nach Typ der Order, Identifier, Instruction oder Literal-Code, läuft Execute sehr unterschiedlich ab.

### 5.3.1 Identifier

Ein Identifier ist eine Zusammenfassung von mehreren Orders. Diese sind in einem Environment definiert. Hierbei handelt es sich um eine einfache Map (Dictionary), **die einen Identifier als string mit einer Liste an Orders verknüpft**. Wenn nun ein Identifier in den Execute Schritt kommt, werden die dazugehörige Liste an Orders auf den Fetch-Stack aus Abschnitt 5.1 gepushed. Beim nächsten Fetch werden nun die zum Identifier gehörenden Orders zurückgegeben. Um Grunde wird der Identifier mit seinen Orders ersetzt.

Environments sind hierbei in einer Linked-List gespeichert. Somit können neue Environments zu dieser Liste hinzugefügt und von der Liste entfernt werden. Das unterste Element der Liste ist hierbei das älteste und das oberste Element das neuste. Bei der Definition eines Identifiers wird dieser 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.

### 5.3.2 Literal-Code

Literal-Code ist der Weg wie der QHS-Compiler Assembly Code generiert. Dieser ist sehr simpel. Wenn Literal-Code in den Execute Schritt gelangt, wird alles was zwischen den SZeichen steht in das Output-Dokument geschrieben.

### 5.3.3 Instructions

Instructions sind die komplexeste Order für den Execute Schritt. Für jede Instruction ist im QHS-Compiler eine Funktion definiert, die ausgeführt wird, wenn diese Instruction in den Execute Schritt gelangt. Diese Funktionen können Variablen im QHS-Compiler speichern, den Order-Queue aktivieren, Identifier definieren und noch viel mehr. Folgend sind ein paar der wichtigsten Instructions aufgelistet. Für manche Instructions ist nach ihrem Namen ein Zeichen als Abkürzung angegeben. Diese Abkürzungen sind in QHS als Identifier definiert, die für diese Instruction stehen. Diese Abkürzungen werden in weiteren QHS Beispielen verwendet.

<b>#enterOrderQueue</b> [	Aktiviert die Order-Queue.
<b>#exitOrderQueue</b> ]	Deaktiviert die Order-Queue.
<b>#assign</b> »	Das erste Element der Order-Queue muss ein Identifier sein. Der Rest der Orders auf Order-Queue wird als Definition für diesen Identifier festgelegt.
<b>#assignToOne</b> ->	Wie #assign, jedoch wird nach dem Identifier nur eine weitere Order von der Order-Queue genommen und als Definition für den Identifier verwendet.
<b>#force</b> !	Die nächste Order wird nach Fetch sofort an Execute weitergegeben. Überspringt Decode und somit die Order-Queue.
<b>#orderEnqueue</b>	Die nächste Order wird sofort der Order-Queue hinzugefügt, auch wenn diese Order Order-Queue-Proof wäre. Execute wird übersprungen.
<b>#orderFrontEnqueue</b>	Ähnlich wie #orderEnqueue. Die Order wird jedoch auf den obersten Platz der Order-Queue gesetzt.
<b>#deepFetch</b>	Wird mit der ersten Order der zweitobersten Liste an Order auf dem Fetch-Stack. Ermöglicht den Zugriff auf den Inputfile innerhalb einer Identifier-Definition.
<b>#pushEnv</b>	Ein neues Environment wird der Environment Linked-List hinzugefügt.
<b>#popEnv</b>	Das neuste Environment der Environment Linked-List wird gelöscht.

Der QHScompiler umfasst **28** Instructions, wobei **5** dieser nur für Debugging des Compilers dienen.

## 5.4 Bringing it all together

Und das war's. Dies ist der gesamte QHScompiler. Im Vergleich zu einem traditionellen Compiler wirkt der QHScompiler fast schon **armselig**. Und dies hat einen einfachen Grund. Der QHScompiler ist zwar **vollendet**, die dazugehörige Sprache QHS jedoch noch lange nicht. Es ist zwar grundsätzlich durch LiteralCode möglich jedes Programm QHS zu schreiben, jedoch handelt es sich dann dabei einfach nur um Assembly Code. Doch der Aufbau des QHScompilers ermöglicht es mithilfe von Identifiern eine komplexere Programmiersprache zu definieren.

### 5.4.1 Shortcuts

Shortcuts like ; and stuff

### 5.4.2 Variablen

Die Umsetzung von Variablen in QHS ist simpel. Zuerst soll die Grösse der Variabel vom Stack abgezogen werden. Dann wird ein für den Identifier ein Variablenamen definiert, der zu der Position der Variabel auf dem Stack zeigt. Mit nur LiteralCode in QHS lässt sich dies wie folgt ausdrücken:

Listing 5.1: THING

```
"sub 4" ;
[ a "[rbp-4]" ] >> /* a = [rbp-4] */
```

```
"add " a ", 5" ;
```

Dieser QHS Code wird zu folgendem compiled:

Listing 5.2: THING

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

Jedoch ist dies noch nicht besonders angenehm. Weiter lässt sich zum Beispiel der Var Identifier definieren, der die Grösse der Variabel oben auf der OrderQueue erwartet.

# Abbildungsverzeichnis

1.1	Schritte, die ein Compiler durchläuft [1]	3
1.2	Schritte, die in dieser Arbeit behandelt werden (Basierend auf Figure 1.1)	3
1.3	Abstract Syntax Tree zum Euklidischen Algorithmus [2]	5
5.1	Zyklus der QHS Compilation	9
5.2	TEMP! Struktur des Fetch-Stacks	10

# Literaturverzeichnis

- [1] Bob Nystrom. A map of territory (mountain.png), 2021. [Online; accessed 2024-08-05].
- [2] Wikipedia. Abstract syntax tree for euclidean algorithm. [Online; accessed 2024-08-05].