Maturaarbeit

Compiler Construction

Fabio Stalder

Betreut durch
Thomas Jampen

12. September 2024

Gymnasium Kirchenfeld
Abteilung MN

Inhaltsverzeichnis

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

1 Introduction

In der Informatik beschreibt *Compiler* ein Programm, das Code aus einer Programmiersprache in eine andere 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 mithilfe einer Assembly Sprache möglich ohne Compiler einem Computer Befehle zu geben, jedoch ist dies aufwendig und nicht gerade simpel. Compiler ermöglichen das Übersetzten von verständlicheren Programmiersprachen zu Assembly und sind daher für die Informatik essentiel. 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 möglichst keinen Raum für Missverständnisse oder Ungenauigkeit lassen. Genauso muss auch ein Compiler exakt und fehlerfrei übersetzten. Neben fehlerfrei muss die Kompilierung auch möglichst schnell sein. Dasselbe gilt natürlich auch für den resultierenden Output-Code. Dieser sollte möglichst optimal generiert werden, um die schlussendliche Ausführungsdauer so kurz wie möglich zu halten. Und falls sich doch einmal ein Fehler im Input-Code befindet, sollten diese verständlich gemeldet werden. Compiler sind also durchaus keine simplen Programme und daher auch bis heute noch ein aktives Forschungsgebiet. In folgendem Text werde ich die Idee, Entwicklung und schlussendliche Auswertung eines von mir erdachten alternativen Ansatzes für den Aufbau eines Compilers beschreiben.

2 Ein traditioneller Compiler

Ein Compiler ist traditionell mit folgendem Schema beschreiben.

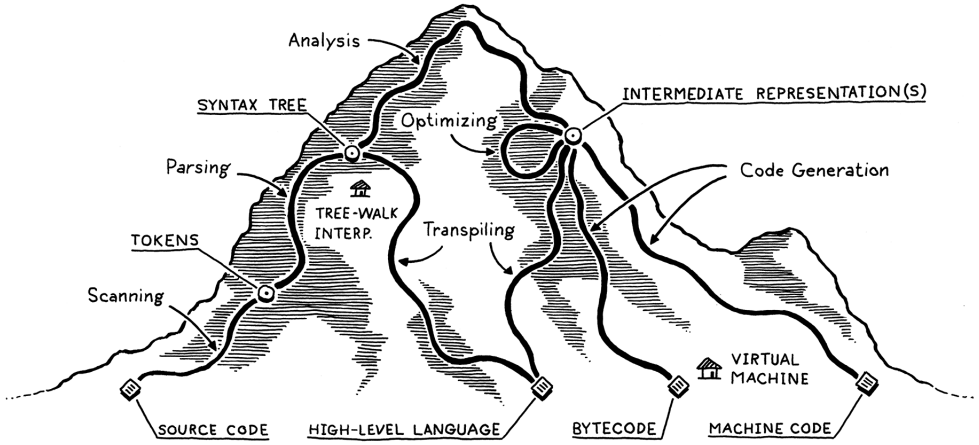

Abbildung 2.1: Schritte, die ein Compiler durchläuft

In dieser Arbeit werde ich mich nur auf die im unteren Schema 2.2 dargestellten Schritte fokussieren.

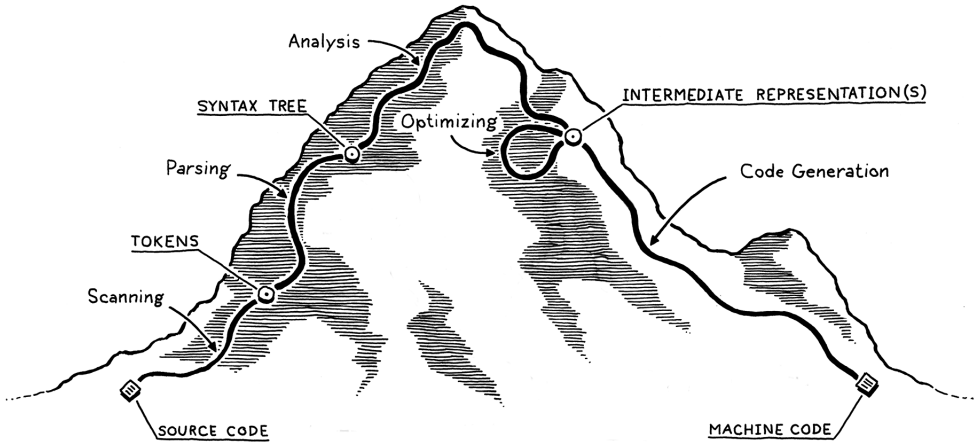

Abbildung 2.2: Schritte, die in dieser Arbeit behandelt werden

2.1 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 noch nicht sofort verständlich. Der erste Schritt beim Kompilieren ist daher die *Lexical Analysis*. Dies wird von einem Teil des Compilers, dem *Lexer*, durchgeführt. Die Aufgabe dieses Lexers ist es den Inputfile zu analysieren und die gefundenen Zeichen in sogenannte *Tokens* zu verwandeln. Diese Tokens sind Datenstrukturen, die der Compiler kennt und mit denen er weiterarbeiten kann.

Als Beispiel:

Listing 2.1: C code vor Lexical Analysis

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

    return 1;
}
```

Listing 2.2: 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 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 2.2 'if' als KeywordToken gesehen wird, lässt sich im Lexer auch bestimmen, dass ein Wort wie 'print' als Keyword angesehen werden soll.

2.2 Syntax Analysis

Nun versteht der Compiler was mit den Zeichen im Inputfile 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. In diesem Schritt für der sogenannte *Parser* die *Syntax Analysis* durch. Hierbei werden die bei der Lexical Analysis gefundenen Tokens ineinander verschachtelt und in einen sogenannten *Abstract Syntax Tree (AST)* überführt.

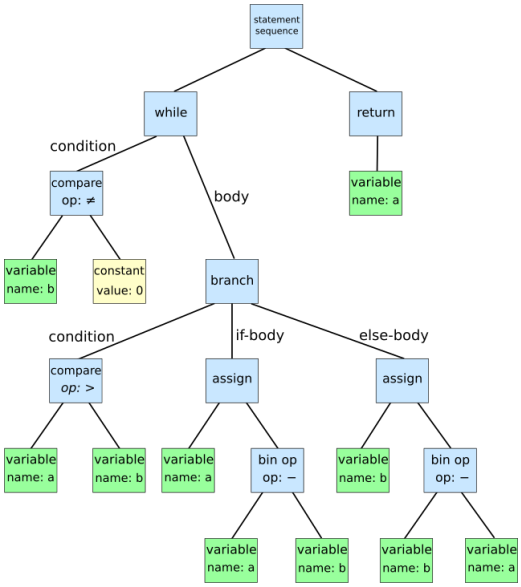

Abbildung 2.3: Abstract Syntax Tree zum Euklidischen Algorithmus

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 2.3 ersichtlich, die Subtraktion über den beiden benötigten Werten im AST.

2.3 Semantic Analysis

Semantik ist die Wissenschaft der Bedeutung von Worten einer Menschensprache. Bei einem Compiler geht es bei der *Semantic Analysis* weniger um Bedeutung und mehr um die Konsistenz von Datentypen. In diesem Schritt der Kompilierung beschäftigt sich der Compiler mit der Korrektheit von Expressions. Wird eine Variable nicht konform ihres Datentyps verwendet, zum Beispiel die Division zweier Strings, 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.

2.4 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. [Code generation types]

2.5 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 viel Verschiedenes bedeuten. Der Output-Code muss so schnell wie möglich ausgeführt werden, Memory sparsam verwenden und am besten auch noch eine möglichst kleine Datei sein. Optimization reicht vom Entfernen der Kommentare und 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 und komplexes Problem, dass hierbei nicht weiter thematisiert werden sollte.

3 Meine Idee

Ein Compiler ist ein äusserst 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. Während meinen Nachforschungen zum Thema Compiler, stiess ich auf den Begriff *Macro Expansion*. Dabei handelt es sich um eine Methode der Code Generation, bei der eine Struktur des AST (z.B. ein If-Statement) mit einem *Macro* ersetzt wird. Bei diesem Macro kann es sich um so ziemlich alles, meist jedoch ein Stuck Assembly oder Object Code, handeln. Somit wird nach und nach der gesamte AST ersetzt, bis nur noch Macros übrig sind. Inspiriert von dieser Methode der Code Generation, überlegte ich mir ein Compiler, der möglichst stark dieses Konzept der Macros verwendet. Ein Compiler, der sowohl Lexical als auch Syntax Analysis überspringen kann und es ermöglicht in einem Programm weitere Macros, und somit eine Art eigene Programmiersprache, selbst zu definieren.

3.1 Vergleich der Compiler

Um die Leistung meiner Idee zu testen, werden folgende Compiler verglichen.

Der *QHScompiler* ist ein von mir nach meiner Idee entwickelter Compiler. Seine genaue Funktionalität wird in Kapitel 4 weiter ausgeführt. Er ist in C++ geschrieben und generiert x86 Assembly mit NASM Syntax aus meiner Sprache QHS.

Der GCC Compiler ist der gebräuchlichste Compiler für die Programmiersprache C. Veröffentlicht im Jahre 1987 wird GCC bis heute weiterentwickelt und ermöglicht heutzutage auch die Kompilierung von C++, Rust, Fortran, usw. [...] <- Add if data is actually useful

Der *THScompiler* repräsentiert in diesem Vergleich einen traditionellen von mir geschrieben Compiler. Im Gegensatz zu GCC ist der THScompiler deutlich simpler und kleiner. Er dient daher als realistische Konkurrenz zum QHScompiler und wird verwendet, um zu testen, wie viel meine Idee taugt. Er folgt dem theoretischen Aufbau eines Compilers und besteht aus Lexer, Parser und Code Generator. Als Parser wird ein Predictive Descent Parser verwendet. Optimization wird nicht separat durchgeführt und ist somit auch sehr schwach. Die Semantic Analysis wird während der Code Generation durchgeführt. Geschrieben ist der Compiler in C++ und liefert x86 Assembly mit NASM Syntax.

3.1.1 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 Kompilierung	Wie lange dauert die Kompilierung von Code?
Benutzerfreundlichkeit	Wie einfach ist die Verwendung des Compilers?
Möglichkeit für Erweiterung	Wie einfach ist die Input-Sprache zu erweitern?

3.1.2 Anforderungen an die Compiler

Ausserdem müssen die Compiler folgende Anforderungen mindestens erfüllen.

Output als Assembly Code	Die Output-Sprache muss Assembly Code sein
C-like Syntax	Die Input-Sprache muss einen C-ähnlichen Syntax aufweisen
Variablen und Funktionen	Lokale und globale Variablen sowie Funktionen müssen unterstützt werden
Branching und Loops	If-Statements sowie While-Loops müssen umsetzbar sein

4 Der QHScompiler

Die QHS Sprache besteht natürlich aus Wörtern. Im Kontext von QHS werden diese Wörter *Orders* genannt. Diese Orders weisen 3 verschiedenen Typen auf. *Identifiers*, *Instructions* und *LiteralCode*. Bei Identifiers handelt es sich um die in Kapitel 3 bereits erwähnten Macros. Instructions sind einfache vorprogrammierte Anweisungen an den QHScompiler und LiteralCode ist Text, der exakt so wie er steht, in den Outputfile geschrieben wird. Wie diese 3 Order Typen genau funktionieren wird in Abschnitt 4.3 ausführlicher erklärt.

Dem QHScompiler steht für die Kompilierung von QHS ein einfacher Zyklus zugrunde, dessen Vorbild der Von-Neumann Zyklus ist.

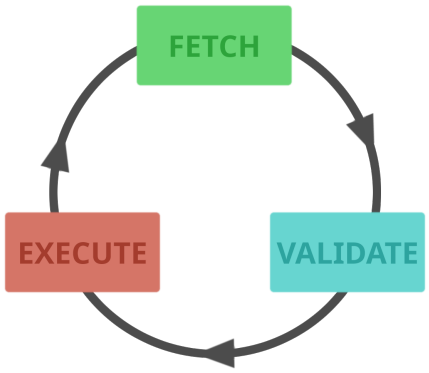

Abbildung 4.1: Zyklus der QHS Kompilierung (QHS-Zyklus)

4.1 Fetch

Der *QHS-Zyklus* beginnt mit 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. Whitespaces dienen als Trennung zwischen zwei Orders und werden ignoriert.

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

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

Auffällig gegenüber einem traditionellen Compiler ist hierbei, dass kaum zwischen Zeichen differenziert

wird. Während die Lexical Analysis traditionell zwischen vielen verschiedenen Tokens unterscheidet, sind für den QHScompiler alle Zeichen bis auf # und "gleichbedeutend.

Bei einem Fetch wird die nächste Order normalerweise vom Inputfile geholt. Es ist jedoch möglich Orders voranzustellen. Diese werden beim nächsten Fetch zuerst gefunden. Dies geschieht mithilfe des *FetchStacks* auf den eine Queue an Orders gelegt werden kann. Beim nächsten Fetch wird zuerst die erste Order der obersten Queue auf dem FetchStack geholt. Wenn eine Queue an Orders komplett gefetched wurde, wird diese vom Stack gelöscht. Der Inputfile befindet sich auf dem letzten Platz des FetchStacks und wird somit nur verwendet, wenn der Stack ansonsten komplett leer ist. Auf den FetchStack kann während jeder der drei Schritte des Zyklus, meist jedoch während Execute, gelegt werden. Während der Laufzeit des QHScompilers könnte der FetchStack folgendermassen aussehen:

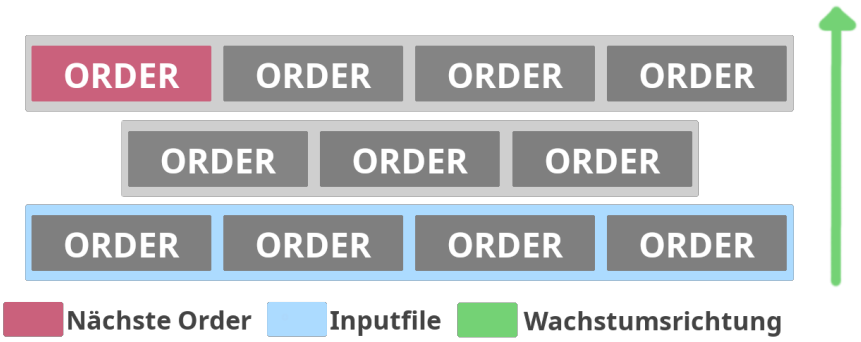

Abbildung 4.2: Struktur des FetchStacks

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

4.2 Validate

Nachdem die nächste Order gefunden wurde, wird diese 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 mithilfe von Instructions, die im Abschnitt 4.3 weiter ausgeführt werden, aktiviert und deaktiviert werden. Wenn nun 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 ausgeführt zu werden auf der OrderQueue gespeichert. Später ist es nun möglich diese Order mithilfe von Instructions, die im Abschnitt 4.3 weiter thematisiert werden, von der OrderQueue zu entfernen und auszuführen. Bestimmte Instructions und Identifiers können jedoch OrderQueue-Proof, also immun gegen die OrderQueue, gemacht werden. Diese werden, auch wenn die OrderQueue aktiv ist, normal an Execute weitergegeben. Dies ist zum Beispiel besonders bei der Instruction, die die OrderQueue wieder deaktiviert, wichtig. Da diese Instruction sonst nicht ausgeführt und somit die OrderQueue nie mehr deaktiviert wird. LiteralCode kann nicht CodeQueue-Proof sein.

MAYBE CODESTACK FIGURE

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

4.3 Execute

Execute ist der letzte Schritt des QHS-Zyklus. 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.

4.3.1 Identifier

Ein Identifier ist eine Zusammenfassung von mehreren Orders. Diese sind in einem *Environment* definiert. Hierbei handelt es sich um eine einfach Map, die einen Identifier mit einer Queue an Orders verknüpft. In anderen Programmiersprachen werden Environments auch als Scope bezeichnet. Wenn nun ein Identifier in den Execute Schritt kommt, werden die dazugehörige Queue an Orders auf den FetchStack aus Abschnitt 4.1 gelegt. 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 Environment der Liste ist hierbei das älteste und das oberste Environment das neuste. Ein neuer Identifiers 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 Literal-Code

Literal-Code ist der Weg wie der QHScompiler 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. Dies ist die einzige Möglichkeit für den QHScompiler Assembly Code zu generieren. Somit könnte nur durch das Ändern von den einzelnen LiteralCode Orders die Output-Sprache des QHScompilers komplett geändert werden.

4.3.3 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 natürlich noch viel mehr. Folgend sind ein paar der wichtigsten Instructions aufgelistet:

#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 erste Order der zweitobersten Liste auf dem FetchStack wird oben auf den FetchStack gesetzt. Ermöglicht den Zugriff auf den Inputfile 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 für Debugging des Compilers dienen.

4.4 Bringing it all together

Und das war's. Dies ist der gesamte QHScompiler. Im Vergleich zu einem traditionellen Compiler wirkt der QHScompiler fast schon zu simpel. Und dies hat einen einfachen Grund. Der QHScompiler ist zwar komplett, die dazugehörige Sprache QHS jedoch noch lange nicht. Es ist zwar grundsätzlich durch LiteralCode möglich jedes Programm zu schreiben und zu kompilieren, 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.

4.4.1 Shortcuts

Um die Leserlichkeit von QHS zu verbessern, werden ein paar Identifiers anstelle der umständlichen Instruction Namen definiert.

`[`	#enterOrderStack
`]`	#exitOrderStack
`>>`	#assign
`->`	#assignToOne
`!`	#force
`?!`	#lightForce
`\n`	Eine neue Zeile im Outputfile

Weiter wird in den Kommentaren innerhalb der Beispiele Pseudo-Code verwendet, um den QHS Code besser zu erklären. Kommentare können über mehrere Zeilen reichen 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 besonders bei längeren Identifier Definitionen zuerst der Identifier Name getrennt von den restlichen Orders der OrderQueue hinzugefügt. Diese Seperation dient lediglich der Leserlichkeit und ist ansonsten nicht nötig.

4.4.2 Identifier Parameters and Return

Mithilfe der #enterOrderQueue und #exitOrderQueue Instructions kann innerhalb eines Identifiers die OrderQueue verwendet werden. Dies ermöglicht eine Art von Parametern und Return für Identifiers. 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.

Listing 4.1: Parameter und Return 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 */
] >>

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


Output
Parameter 1 : Parameter 2
Return
```

4.4.3 Variablen

Die Umsetzung von Variablen in QHS ist simpel. 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 zu der Position der Variable auf dem Stack zeigt. Mit nur LiteralCode in QHS lässt sich dies wie folgt ausdrücken:

Listing 4.2: Definition einer Variable mit viel LiteralCode

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

"add " a ", 5"
```

Output

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

Jedoch ist dies noch nicht besonders angenehm. Weiter lässt sich zum Beispiel ein *var* Identifier definieren, der die Grösse der Variable als Argument über die OrderQueue annimmt. Um die in vielen Programmiersprachen geläufige Syntax einer Variable beizubehalten, wird der Name der Variable mithilfe der #deepFetch Instruction beschafft.

Listing 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
```

Ganz so richtig funktioniert dies aber noch nicht. Momentan erhält jede Variable die Addresse rbp-4 und somit überschreiben sich die Variablen gegenseitig. Der momentane rbp-Offset muss also gespeichert und erhöht werden. Hierzu wird bereits am Anfang des Programms ein Identifier rbpOffset als 0 definiert. Mithilfe der #addToIdentifier Instruction, lässt sich daraufhin rbpOffset erhöhen. Dies kann folgendermassen aussehen:

Listing 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 Identifier zusammenfassen. Dies wäre passenderweise die bekannte Bezeichnung für den Typen der Variable.

Listing 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
```

So sieht eine Variable genau so aus, wie es in anderen Programmiersprachen gebräuchlich ist.

4.4.4 Funktionen

Funktionen sind im Vergleich zu Variablen deutlich komplizierter. Hierbei sollen zwei der Problematiken an Funktionen behandelt werden.

Zum Schluss sollte eine Funktionsdefinition wie folgt aussehen:

Listing 4.6: Ziel für die Definition einer Funktion in QHS

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

Hier lässt sich bereits ein erstes Problem feststellen. Im vorherigen Abschnitt 4.4.3 wurde der *int* Identifier für die Definition einer Variable verwendet. Das int aus Beispiel 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. Dies 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 Problem jedoch 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 nun eine Klammer als Identifier für eine Funktionsdefinition gesetzt, sowie ein Semikolon für die Definition einer Variable. Befindet sich nach dem Namen also 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 könnte dann wie folgt aussehen:

Listing 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 eine die Definition einer Variable, sollen 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 hierzu liegt im Umdefinieren des *definition* Identifiers. Dieser ist momentan für sowohl für Variable als auch Funktionsdefinitionen 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 umsetzten:

Listing 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 hierbei gleich wie der *var* Identifier aus Abschnitt 4.4.3. Jedoch wird anstelle von *rbpOffset* ein neuer *paramOffset* Identifier verwendet.

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

Listing 4.9: Finale Definition einer Funktion in QHS

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

Output

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

Mithilfe von DelayedExecute und TempAssign lassen sich also auch syntaktisch komplexen Code problemlos in QHS definieren und ausführen. **Wie das Callen von Funktionen aussieht, soll hierbei nicht weiter betrachtet werden.**

5 Auswertung

Wie bereits in Kapitel 3 beschrieben wurde, sollen drei Compiler QHScompiler, THScompiler und GCC sowie deren dazugehörigen Sprachen QHS, THS und C verglichen werden. Diese werden in Geschwindigkeit der Kompilierung, Geschwindigkeit eines kompilierten Programmes, 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.

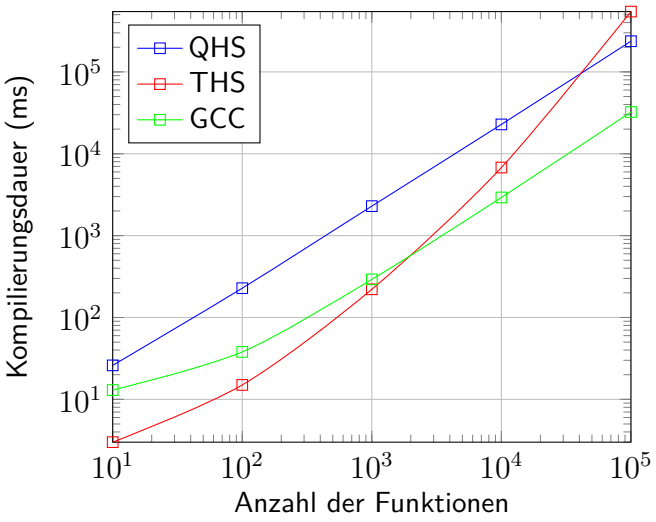

Abbildung 5.1: Vergleich der Kompilierungsdauer mit Log-Log Skalen

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 eines Programmes

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.

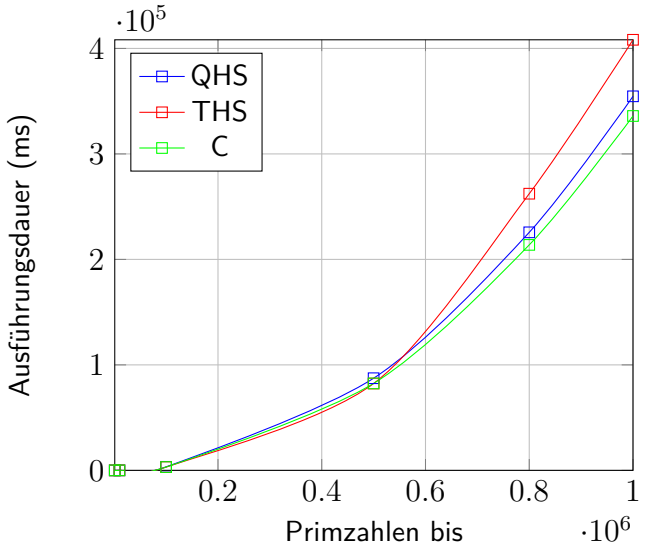

Abbildung 5.2: Vergleich der Ausführungsdauer

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.

Listing 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.

Listing 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.

Listing 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: EMPTY
[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: EMPTY
[ERROR] Expected LiteralCode for #literalToIdentifier, got: EMPTY
[ERROR] Expected LiteralCode for #literalToIdentifier, got: EMPTY
```

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 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 QHS-compiler.

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.

Der QHScompiler ist hierbei jedoch noch etwas interessanter. 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. Es lässt sich also im Grunde eine komplett andere Sprache als QHS ohne jegliche Änderung am QHScompiler definieren. Jedoch ist dies nicht besonders intuitiv und sehr fehleranfällig.

Der QHScompiler ermöglicht einem also grundsätzlich mehr Freiheit. (...)

6 Fazit

Der QHScompiler hat im Vergleich 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 zum Verwenden. Als einziger Vorteil lässt sich seine Möglichkeit zur Erweiterung sehen. Daher lässt sich klar sagen, zum Compiler ist der QHScompiler nicht besonders gut geeignet. Jedoch ist deswegen der QHScompiler nicht gleich nutzlos. Mit etwas Optimierung könnte er nämlich immer noch als praktisches Tool dienen. Und zwar fürs Schreiben von Assembly Code. Man könnte mit LiteralCode weiterhin den grössten Teil des Codes normal in Assembly schreiben, sich häufig wiederholende Code Stücke jedoch mit Identifiern abkürzen. Besonders für spezialisierte Prozessor Architekuren mit spezifischen Instructionsets könnte der QHScompiler eine einfachere Alternative zu einem kompletten Compiler darstellen. Somit ist der QHScompiler zwar keine bahnbrechende Idee, die die modernen Compiler in den Schatten stellt, aber zumindest ein kleines nützliches Tool, dass ich vielleicht das ein oder andere Mal noch verwenden werde.

Abbildungsverzeichnis

Listings

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].