Maturaarbeit

Compiler Construction

Fabio Stalder

Betreut durch
Thomas Jampen

2. September 2024

Gymnasium Kirchenfeld
Abteilung MN

Inhaltsverzeichnis

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. Jedoch unterscheidet sich ein Compiler grundsätzlich von Übersetzern in der Erwartungshaltung, die an sie gestellt wird. Menschensprache ist sehr komplex und [...]

I have Idea

2 Traditioneller Compiler

Ein Compiler ist traditionell nach folgendem Schema aufgebaut.

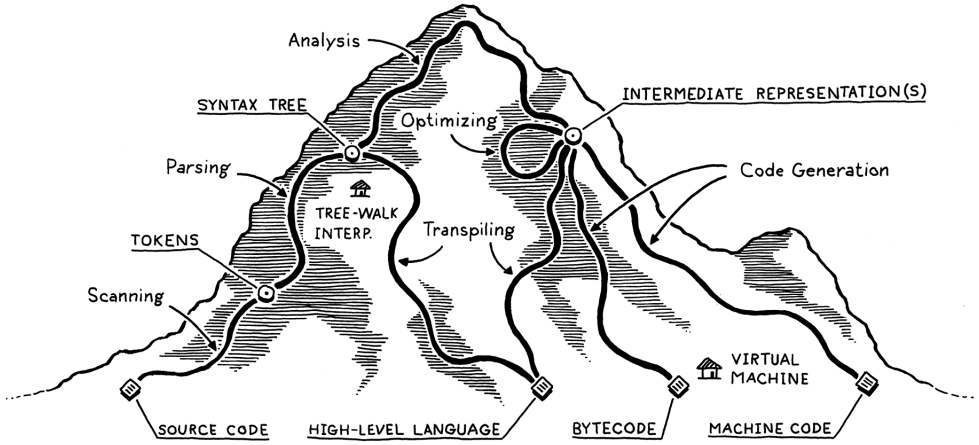

Abbildung 2.1: Schritte, die ein Compiler durchläuft [1]

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

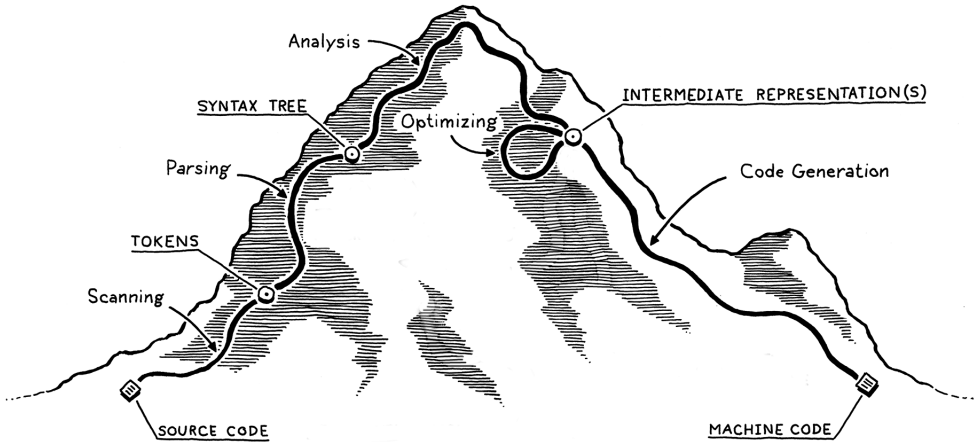

Abbildung 2.2: Schritte, die in dieser Arbeit behandelt werden (Basierend auf Figure 2.1)

2.1 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 2.1: 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 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 Input File gemeint ist, jedoch fehlt noch etwas bis tatsächlich in einer 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. [...]

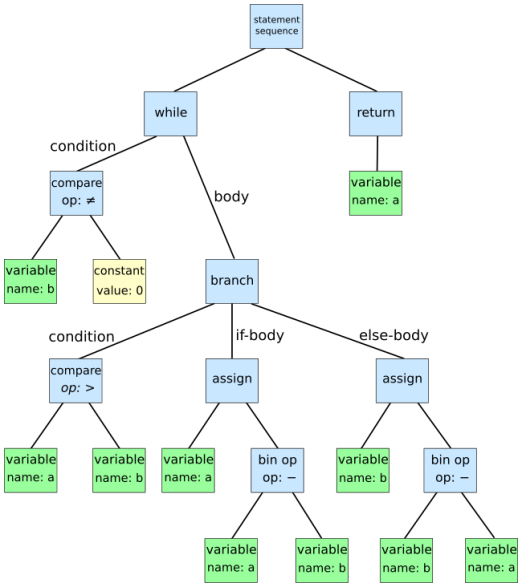

Abbildung 2.3: Abstract Syntax Tree zum Euklidischen Algorithmus [2]

2.3 Semantic Analysis

Semantik ist die Wissenschaft der Bedeutung von Worten einer Menschensprache. So ähnlich ist es auch bei Programmiersprachen, jedoch geht hierbei viel weniger um Bedeutung und eher um den Datatype. In diesem Schritt der Compilation beschäftigt sich der Compiler mit der Validität von Expressions. Unbekannte Variablen oder Funktionen werden in diesem Schritt abgefangen. Weiter wird der Datatype einer Node an diese angebunden. Gegebenenfalls kann auch ein Implicit-Cast, also ein impliziter Wechsel des Datatypes hinzugefügt werden. So geben zum Beispiel manche Programmiersprache bei der Division zweier Integers eine Float zurück. Wird eine Variable nicht konform ihres Datatypes verwendet, zum Beispiel die Division zweier Strings, wird dies ebenfalls während der Semantic Analysis entdeckt und gemeldet.

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 Compilieren, trotzdem wurde eine wichtige Aufgabe des Compilers noch nicht betrachtet. Optimization ist ein Sache die zwischen jedem der genannten Schritte geschiet und dies häufig mehrmals. 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 und komplexes Problem, dass hierbei nicht weiter thematisiert werden sollte.

3 Meine Idee

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. Während meinen **Nachforschungen** zum Thema Compiler, stiess ich auf den Begriff *Macro Expansion*. Dabei handelt es sich um eine Methode zur 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. Ein Compiler, der 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 der Compiler verglichen.

Der *QHS Compiler* 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 nach 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 Compilation von C++, Rust, Fortran, usw. [...] <- Add if data is actually useful

Der *THS Compiler* repräsentiert in diesem Vergleich einen traditionellen von mir geschrieben Compiler. Im Gegensatz zu GCC ist der THS Compiler deutlich simpler und kleiner. Er dient daher als realistische Konkurrenz zum QHS Compiler 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. 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.

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 Compilation	Wie lange dauert die Compilation 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

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-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 QHS Compiler

Die QHS Sprache besteht natürlich aus Wörtern. Im Kontext von QHS werden diese Wörter im folgenden als Orders benannt. Diese Orders weisen 3 verschiedenen Typen auf. Identifiers, Instructions und Literal-Code. 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 Output-File geschrieben wird. Wie diese 3 Orders genau funktionieren wird in Abschnitt 4.3 ausführlicher erklärt.

Der Compilation von QHS steht ein einfacher Zyklus zugrunde, dessen Vorbild der Von-Neumann Zyklus ist.

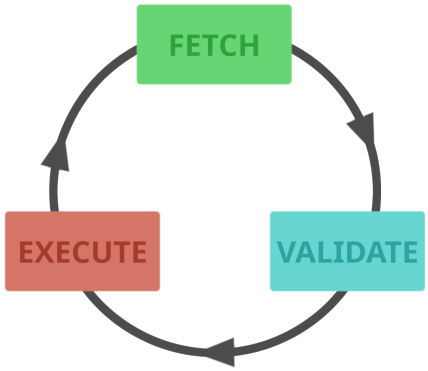

Abbildung 4.1: Zyklus der QHS Compilation

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 "gleich.

Es ist hierbei möglich bestimmte Orders voran 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. Auf diesen Fetch-Stack kann während jeder der drei Schritte des Zyklus, meist jedoch während Execute, gepushed werden. Wie der Name Stack schon sagt, funktioniert der Fetch-Stack mit Last-In First-Out. Die Hauptanwendung des Fetch-Stacks wird im Abschnitt 4.3 ausgeführt.

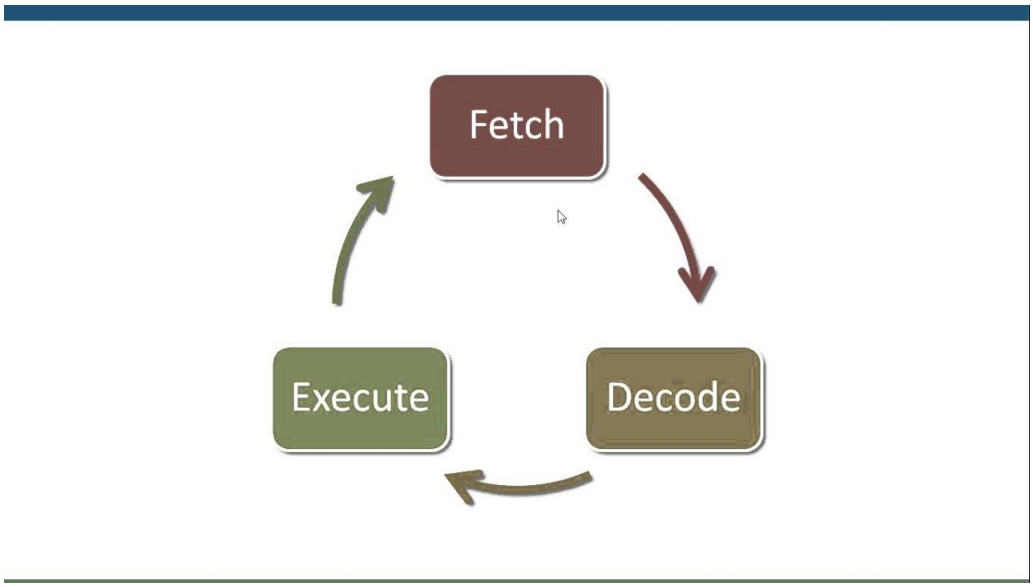

Abbildung 4.2: TEMP! Struktur des Fetch-Stacks

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.

4.2 Validate

Nachdem eine Order gefetched wurde, wird diese an Validate weitergegeben. Während dem Validate Schritt kommt die OrderQueue ins Spiel. Hierbei handelt es sich um eine Liste an Orders, der Form First-In First-Out. Die Anwendung Aufgabe der OrderQueue ist das Speichern und spätere Ausführen von Orders. Die OrderQueue kann mit Hilfe 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 Zyklus beginnt von neuem bei Fetch. Die Order wurde ohne ausgeführt zu werden auf der OrderQueue gespeichert. Später ist es nun möglich diese Order mit Hilfe 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 Code-Queue-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 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.

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 (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 4.1 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.

4.3.2 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.

4.3.3 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 Variabeln im QHS-Compiler speichern, den OrderQueue aktivieren, Identifier definieren und noch viel mehr. Folgend sind ein paar der wichtigsten Instructions aufgelistet.

#enterOrderQueue	Aktiviert die OrderQueue.
#exitOrderQueue	Deaktiviert die OrderQueue.
#assign	Das erste Element der OrderQueue muss ein Identifier sein. Der Rest der Orders auf 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.
#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	Wird mit der ersten Order der zweitobersten Liste an Order auf dem Fetch-Stack. Ermöglicht den Zugriff auf den Inputfile innerhalb einer Identifier-Definition.
#pushEnv	Ein neues Environment wird der Environment Linked-List hinzugefügt.
#popEnv	Das neuste Environment der Environment Linked-List wird gelöscht.

ADD put in front and addIdentifier and lightForce

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 **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.

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 Output-File

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.

4.4.2 Identifier Parameters and Return

Parameter and Return through OrderQueue or PutInFront lateArg» ?

4.4.3 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 Variabelnamen definiert, der zu der Position der Variabel auf dem Stack zeigt. Mit nur LiteralCode in QHS lässt sich dies wie folgt ausdrücken:

Listing 4.1: Beispiel einer Variabel in QHS

```
"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 Variabel als Argument annimmt OrderQueue. Um die in vielen Programmiersprachen geläufige Syntax einer Variabel Definition beizubehalten, wird der Variabel Name mithilfe der #deepFetch Instruction beschafft.

Listing 4.2: Definition einer Variable mit var Identifier

```
[ var ]
[
    #putInFront 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 wiefolgt aussehen.

Listing 4.3: Definition einer Variable mit rbpOffset

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

[ var ]
[
    #putInFront 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 Variabel Typen.

Listing 4.4: Definition einer Variable mit int Identifier

```
(...)

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

int a
int var 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 Variabel Definition genau so aus, wie es in anderen Programmiersprachen gebräuchlich ist. Der Shortcut ; ist hierbei optional.

4.4.4 Funktionen

Funktionen sind im Vergleich zu Variablen deutlich komplizierter. **Daher soll hierbei nur auf zwei der Problematiken an Funktionen behandelt werden.**

Zum Schluss sollte eine Funktionsdefinition wie folgt aussehen:

Listing 4.5: Ziel für die Definition einer Funktion in QHS

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

Output

```
foo:
(...)
```

Hier lässt sich bereits ein erstes Problem feststellen. Im vorherigen Abschnitt 4.4.3 wurde der Int Identifier für eine Variabel Definition verwendet. Das int aus Beispiel 4.5 würde vom QHScompiler also als eine Variabel Definition verstanden werden. Der Unterschied zwischen Funktions und Variabel Definition 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 Variabel 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 Variabel als auch eine Funktionsdefinition vorzubereiten, aber keine der beiden bereits auszuführen. Weiter wird nun eine Klammer als Identifier für eine Funktionsdefinition definiert. Sowie ein Semikolon als Variabeldefinition. Befindet sich nach dem **Namen** also eine Klammer, wird eine Funktionsdefinition ausgeführt. Ist dort aber ein Semikolon wird eine Variabeldefinition durchgeführt. Dieses Konzept wird weiterführend als DelayedExecute beschrieben. Das ganze könnte dann wie folgt aussehen:

Listing 4.6: Implementation eines DelayedExecute für Definitionen

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

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

[ definition ]
[
    #putInFront 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 Variabeldefinition, sollen jedoch vom QHScompiler anders ausgeführt werden. Erstens sollte bei einer Parameterdefiniton nicht der LiteralCode zur Subtraktion vom RSP hinzugefügt werden. Zweitens verwendet eine Parameterdefintion einen anderen RBP-Offset. Die Lösung hierzu liegt in einer Umdefinition des Definition Identifiers. Dieser ist momentan für sowohl für Variabel 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 4.7: Implementation eines TempAssigns für Parameter Definitionen

```
[ function ]
[
    #pushEnv

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

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

    [ definition paramDefinition ] /* definition = paramDefinition */

    #popEnv     /* neu Definition 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 der Funktionsdefiniton, der Funktionsbody. Dieser ist vergleichsweise simpel. Die beiden geschwungenen Klammern werden ganz einfach zu einem leeren Identifier definiert und somit einfach ignoriert. Der gesammte Code innerhalb des Bodies wird nun einfach ganz normal vom QHScompiler ausgeführt und an den Output Assembly File angehenkt. **Wie das Callen von Funktionen aussieht, soll hierbei nicht weiter betrachtet werden.**

Mithilfe von DelayedExecute und TempAssign lassen sich also auch syntaktisch komplexen Code problemlos in QHS definieren und ausführen.

Abbildungsverzeichnis

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].