Maturaarbeit

Compiler Construction

Fabio Stalder

Betreut durch
Thomas Jampen

7. August 2024

Gymnasium Kirchenfeld
Abteilung MN

Inhaltsverzeichnis

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.

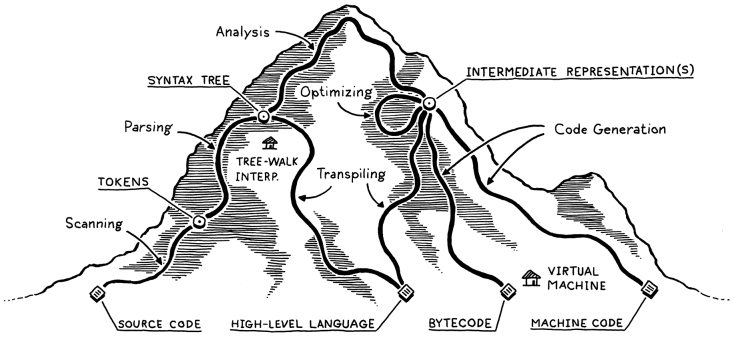

Abbildung 1.1: Schritte, die ein Compiler durchläuft [1]

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

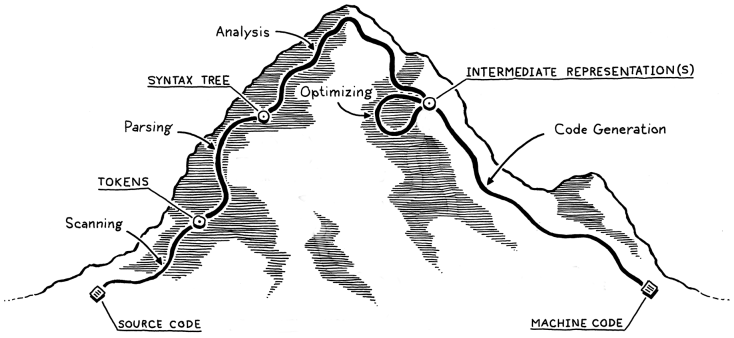

Abbildung 1.2: Schritte, die in dieser Arbeit behandelt werden (Basierend auf Figure 1.1)

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 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.

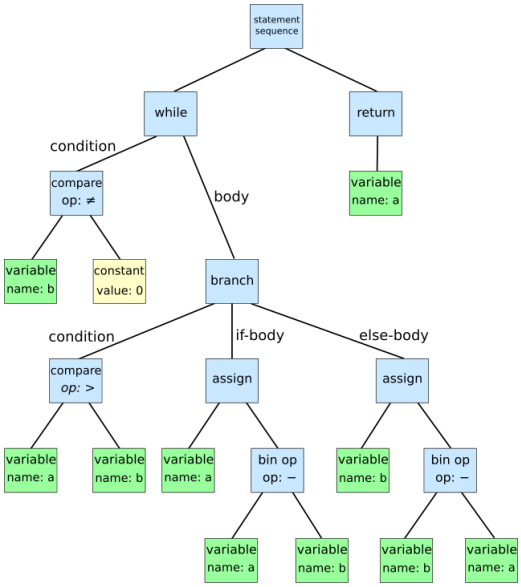

Abbildung 1.3: Abstract Syntax Tree zum Euklidischen Algorithmus [2]

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. [...]

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 ein Sprache die zwischen jedem der genannten Schritte geschiet. 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.

Geschwindikeit des Output-Codes	Wie schnell wird der Output-Code ausgeführt?
Geschwindikeit 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 Schreiben der Compiler

4.1 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.

4.2 QHS Compiler

Der QHS Compiler beginnt wie ein theoretischer Compiler mit Lexical Analysis. Die Syntax Analysis wird jedoch übersprungen. Genauso wie der THS Compiler ist auch der QHS Compiler in C++ geschrieben und generiert x86 Assembly nach NASM Syntax.

4.2.1 Lexical Analysis

Der QHS Compiler Lexer Unterscheidet nur zwischen 4 Tokens: Whitespaces, Identifiers, Instructions und LiteralCode. **Die** Lexical Grammar von THS definiert die Tokens wie folgt:

<whitespace>	SPACE | NEWLINE | TAB
<identiferChar>	[^# "<whitespace>]
<identifier>	<identiferChar>*
<instruction>	# <identiferChar>*
<literalCode>	".*"

4.2.2 Code Generation

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].