Seminararbeit

64-852-S Integriertes Seminar Effiziente Programmierung

DIY Compilerbau

Hauke Stieler

30. April 2020

Fachbereich Informatik
WR – Wissenschaftliches Rechnen
Integriertes Seminar Effiziente Programmierung

Inhaltsverzeichnis

1 Einführung

Diese Seminararbeit behandelt im Wesentlichen die Funktionsweise von Compilern, stellt die Compiler-Pipeline vor und geht besonders auf das Compiler-Framework LLVM ein.

Ein *Compiler* (im Deutschen häufig *Übersetzer*) ist ein Programm, welches rohen Quelltext entgegen nimmt und diesen in eine oder mehrere Binärdateien übersetzt. Es gibt dabei verschiedene Arten von Binärdateien [Rog, S. 8–11]:

- Direkt ausführbare Dateien (häufig *executables* genannt) können direkt auf der vorher bestimmten Computerarchitektur (z.B. AMD64) ausgeführt werden
- *Object files* (zu Deutsch *Objektdateien*) können mit externen Bibliotheken und mittels eines *linkers* zu einem ausführbaren Programm verbunden werden (*statisches linken*), was aber auch erst zur Laufzeit passieren kann (*dynamisches linken*)
- Dateien mit *Zwischencode* (im englischen *intermediate representation* oder kurz *IR*) können durch Nutzung einer Laufzeitumgebung zu Ausführung gebracht werden, was beispielsweise bei Java der Fall ist

Wie schon erwähnt beschäftigt sich diese Arbeit besonders mit dem LLVM-Framework, wodurch viel auf die Erzeugung von Zwischencode eingegangen wird. Am Ende dieser Arbeit zeige ich zudem einen Beispiel-Compiler, welcher LLVM-Zwischencode nutzt um darauf eine object-file zu erstellen.

1.1 LLVM

Im weiteren Verlauf wird häufiger auf LLVM eingegangen, die Prinzipien sind jedoch auch in anderen Compilern (wie z.B. der GCC Compiler) vorhanden.

LLVM ist eine Sammlung von Werkzeugen um unter anderem Compiler zu bauen, welche von der Zielarchitektur unabhängig sind. [Wik] Damit ist LLVM nur der Unterbau, welcher eine Zwischensprache entgegen nimmt, diesen Code analysiert und optimiert und am Ende beispielsweise eine Objektdatei erzeugt. Wichtige Aspekte von LLVM bei der Kompilierung sind verschiedene Analyse und Optimierungsverfahren, sowie die die strikte Nutzung von SSA (static single assignment) Anweisungen.

Bekannte Nutzungen von LLVM finden sich zum Beispiel im CLang C-Compiler oder das Analysewerkzeug KLEE.

Beim Programmieren eines eigenen Compilers bindet man dabei im eigenen C++ Code LLVM als ganz normale Bibliothek ein, welcher man Objekte LLVM-eigener Klassen übergibt, die zum Beispiel Variablenzuweisung modellieren. Details dazu jedoch am Ende im Abschnitt 3.

2 Funktionsweise eines Compilers

Compiler werden in der Regel ein Form einer Pipeline aufgebaut, sie bestehen also aus verschiedenen Schritten, welche Daten (in diesem Fall Code) entgegen nehmen, verarbeiten (analysieren, transformieren, optimieren, etc.) und an den nächsten Schritt weiter geben.

Bei einer Compiler-Pipeline wird der rohe Quelltext zunächst analysiert und somit auf Korrektheit überprüft. Am Ende steht dann entsprechend die Code-Erzeugung, also das erzeugen einer ausführbaren Datei oder dergleichen.

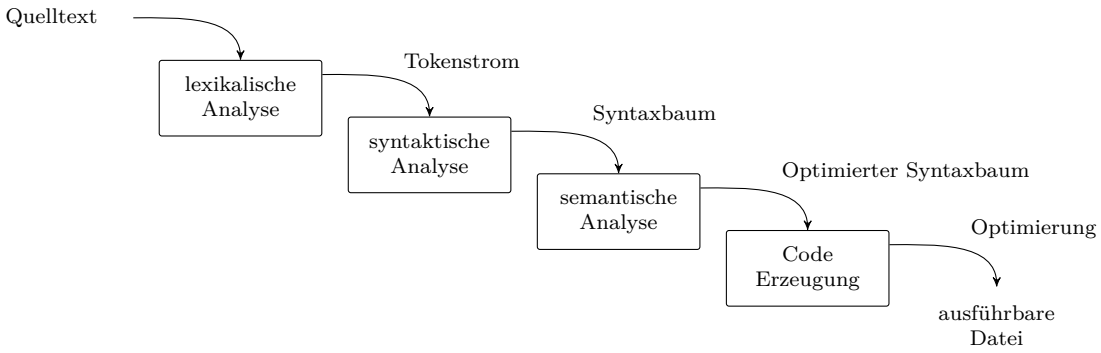

Abbildung 1: Phasen eines Compilers. Basierend auf [Rog, S. 13]

2.1 Pre-processing

Dieser Schritt ist streng genommen nicht Teil des Compilers, zumindest bei GCC und LLVM befindet sich dieser Schritt nicht im eigentlichen Compiler. Im Pre-processing werden meist Makros aufgelöst und bedingte Kompilierung von Code vorgenommen.

Makros sind meist eingebettete Konstanten oder kleinere Codefragmente, welche an vielen Stellen wiederverwendet werden. Hierbei findet lediglich, ohne weitere Analyse auf Korrektheit, eine simple Ersetzung der Makros mit deren Definition statt.

Bedingte Kompilierung ist ein Hilfsmittel um abhängig von einer Bedingung Code zu Kompilieren oder eben nicht zu Kompilieren (siehe Listing 1).

```
1 // Makro:
2 #define BUF_SIZE 64
3
4 // Bedingte Kompilierung:
```

```
5 #ifdef DEBUG
6     #define log_dbg(fmt, ...) printf("DEBUG: " fmt "\n", ##__VA_ARGS__)
7 #else
8     #define log_dbg(fmt, ...)
9 #endif
```

Listing 1: Beispiele für Makros und bedingte Kompilierung in C.

Die Bedingte Kompilierung ist abhängig vom Vorhanden sein des `DEBUG` Makros. Ist dieses zur Übersetzungszeit vorhanden/gesetzt, wird jedes Vorkommen von `log_dbg` durch den angegebenen `printf`-Ausdruck ersetzt. Ist das Makro hingegen nicht definiert, wird `log_dbg` durch nichts ersetzt, effektiv werden also in diesem Fall alle `log_dbg` gelöscht.

Setzen kann man ein solches Makro beispielsweise über Parameter am Compiler:
`$ clang -D DEBUG main.c -o main`

2.2 Lexikalische Analyse

Der erste Schritt des eigentlichen Compilers ist die *lexikalische Analyse* und wird von einem Teil des Compilers übernommen, den man *Lexer* oder auch *Scanner* nennt [Rog, S. 14]. Das Ergebnis der lexikalischen Analyse ist das übersetzen des Quelltextes in eine Folge von sogenannten *Token*.

Ein Token ist eine Datenstruktur, welche vom Quelltext abstrahiert und Zeichen oder Wörter des Quelltextes in Kategorien einteilt. Häufig verwendete Kategorien sind dabei: *Bezeichner* (englisch: *identifier*), *Schlüsselwort* (*keyword*), *Literal* (*literal*), *Separator* (*separator*) und *Operator* (*operator*).

Der Quelltext wird dabei Zeichen für Zeichen eingelesen und in Token aufgeteilt. Der Lexer übernimmt dabei keinerlei Überprüfung auf Korrektheit. Ungültiger Quelltext wie zum Beispiel `int int = int;` wird problemlos in Token übersetzt, da die Überprüfung auf die syntaktische Korrektheit entsprechend in der *syntaktischen Analyse* geschieht, was in Abschnitt 2.4 behandelt wird.

```
1 int i = j * 2;
2
3 if ( i < 100 ) {
4 return 0;
5 }
```

Listing 2: Roher Quelltext.

```
1 int i = j * 2;
2
3 if ( i < 100 ) {
4 return 0;
5 }
```

Listing 3: Quelltext eingeteilt in Token.

Für den weiteren Verlauf kann es zwei Mögliche Wege geben [Rog, S. 15]: Entweder die Token werden als Strom (stream) weitergegeben, sodass die syntaktische Analyse aktiv nach und nach die Token einliest. Oder aber die Token werden gebündelt an die syntaktische Analyse weitergegeben, zum Beispiel als Array, Liste oder Datei. Welche Strategie verfolgt wird, ist allerdings eine reine Implementationsentscheidung.

2.3 Syntaktische Analyse

Die erzeugten Token werden nun an den nächsten Schritt übergeben, dem sogenannten *Parser*, welcher die *syntaktische Analyse* durchführt [Rog, S. 17].

Das Ergebnis des Parsers besteht aus mehreren Teilen, der wesentliche ist aber der *abstrakte Syntaxbaum* (auch: *Parsebaum* oder *Syntaxbaum*; Englisch: *abstract syntax tree* oder kurz *AST*). Dieser bildet nun auch weitere Aspekte mit ab, wie etwa die Schachtelung von Codeblöcken/Ausdrücken und auch die Schachtelung innerhalb von Ausdrücken. Um eine solche Umwandlung in eine Baumstruktur vornehmen zu können, muss der Quelltext einem genauen Regelwerk dienen, welches die Syntax der Sprache definiert. Daher werden in diesem Teil des Compilers auch syntaktische Fehler gefunden und berichtet.

2.3.1 Backus-Naur Form

Wie oben bereits erwähnt, folgen Programmiersprachen einer genauen Spezifikation für die Syntax, welche eine Menge von Regeln enthält. Solche Regeln definieren zum Beispiel, wie eine korrekte Variablendeklaration aussehen muss und welche Sprachbestandteile enthalten sein dürfen oder müssen.

Zur Notation solcher Regeln (oder genauer *Produktionsregeln*) wird häufig die *Backus-Naur Form* (oder kurz *BNF*) verwendet.

```
<statement> ::= <labeled-statement>
        | <expression-statement>
        ...
        | <jump-statement>

<jump-statement> ::= goto <identifier> ;
        | continue ;
        | break ;
        | return {<expression>}? ;
```

Abbildung 2: Auszug aus einer möglichen BNF für `C` [Gup]

Die BNF für das Beispiel in Abbildung 2 definiert unter anderem, wie ein `jump-statement` aufgebaut ist. Ein solcher Ausdruck besteht hier aus einem `goto` gefolgt von einem `identifier` (welches an anderer Stelle in der BNF definiert wird) und einem Semikolon. Senkrechte Striche markieren Alternativen, so wäre ein `break` gefolgt von einem Semikolon ebenfalls ein gültiges jump-statement.

Die festen Begriffe, wie zum Beispiel `goto` oder `continue`, werden *Terminalsymbol* genannt, da es sich um die tatsächlichen Worte handelt und für diese keine weiteren Regeln existieren. Eingeklammerte Element, wie zum Beispiel `<identifier>`, werden *non Terminalsymbol* genannt, da es für sie eine weiterführende Regeln gibt.

2.3.2 Aufbau des AST

Ist eine Spezifikation der Syntax vorhanden, zum Beispiel in Form einer BNF, so kann man anfangen die Token einzulesen und den AST zu erstellen. Der AST besteht aus verschiedenen Knoten, welche Sprachelemente, wie zum Beispiel eine Variablendeklaration, abbilden.

Um herauszufinden bei welcher Abfolge von Token welcher Knoten im AST korrekt ist, müssen gegebenenfalls mehrere Token eingelesen werden. Die Implementation besteht dabei im Grunde exakt aus den BNF Regeln.

Ein Beispiel:
Angenommen `int i;` und `int i=0;` sind gültige Ausdrücke und der Parser fängt an die Token eines solchen Ausdrucks einzulesen. Zuerst wird der Token für `int` und dann der für `i` eingelesen. Der nächste Token darf dann entweder ein `;` (ende der Deklaration) oder ein `=` (direkt anschließende Wertzuweisung) sein. Nachdem also drei Token eingelesen wurden ist klar, um welchen Sprachbestandteil (Deklaration oder Deklaration mit Zuweisung) es sich handelt und welcher AST-Knoten erzeugt werden muss.

Im Wesentlichen wird im Parser also eine Art pattern-matching der Token auf die Regeln betrieben um den AST zu generieren. Wie ein vereinfachter AST aussehen kann, sieht man in Abbildung 3.

2.4 Semantische Analyse

Nachdem die syntaktische Analyse fertiggestellt wurde, wird der AST der semantischen Analyse übergeben [Wat17, S. 141]. Wie der Name bereits sagt, wird hier die Semantik des Codes überprüft. Dabei ist jedoch nicht gemeint, dass etwa Unit-Tests sicher stellen, dass die Programmlogik korrekt ist, sondern es wird sichergestellt, dass Ausdrücke beispielsweise ein definiertes Ergebnis liefern.

Ein Beispiel: Die Addition zweier Ganzzahlen ist erlaubt, aber was passiert bei der Addi-

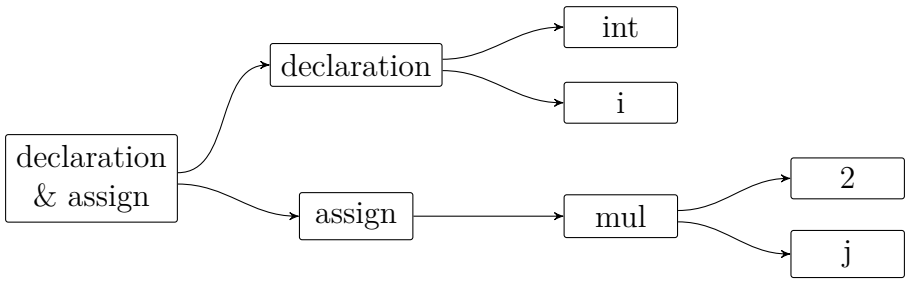

Abbildung 3: Beispielhafter AST für den Ausdruck „`int i = 2 * j;`“.

tion einer Ganzzahl und einer Gleitkommazahl? Ist dieses Verhalten nicht entsprechend definiert, ist eine solche Operation ungültig und führt zu einem Fehler.

Wichtig ist hierbei das Erzeugen einer sogenannten *Symboltabelle*, welche Informationen über identifier, wie zum Beispiel Typinformationen, speichert.

2.4.1 Symboltabelle und annotierter AST

Aus verschiedenen Gründen werden sich Variablen- oder auch Funktionsnamen (allgemein sogenannte *identifier)* in einer Symboltabelle gemerkt [Rog, S. 19, 20]. Gespeichert wird dabei nicht nur die Tatsache, dass gewisse identifier Definiert wurden und somit an anderer Stelle verwendet werden können, sondern es werden auch Metadaten zu den einzelnen identifiern gespeichert. Die Metadaten können neben dem Namen beispielsweise auch Typinformationen oder den Sichtbarkeitsbereich (*scope*) enthalten [Wat17, S. 148].

Beim Erstellen der Symboltabelle können ebenfalls weitere Überprüfungen stattfinden, sodass z.B. identifier nicht mehrfach deklariert werden dürfen.

Diese Informationen werden zunächst genutzt um den AST um weitere Informationen anzureichern, man spricht dabei häufig von annotieren, wodurch ein *annotierter AST* entsteht. In einem solchen annotierten AST kann man dann beispielsweise weitere Anpassungen vornehmen, wie beispielsweise implizite Typumwandlungen.

2.5 Synthese und Zwischencode

Dies ist im Wesentlichen der letzte Schritt eines Compilers, denn hier wird das Ergebnis generiert, welches meist aus einem Zwischencode besteht. Es gibt dabei nicht *den* Zwischencode, sondern diverse verschiedene Arten und Namen für Zwischencode. Java benutzt den begriff *bytecode* für den Code, welcher von der JVM (Java virtual machine) zur Laufzeit compiliert und zur Ausführung gebracht wird. Der Compiler GCC benutzt diverse Zwischencodes, wie etwa GENERIC, GIMPLE und RTL [Nov]. Aus letzterem

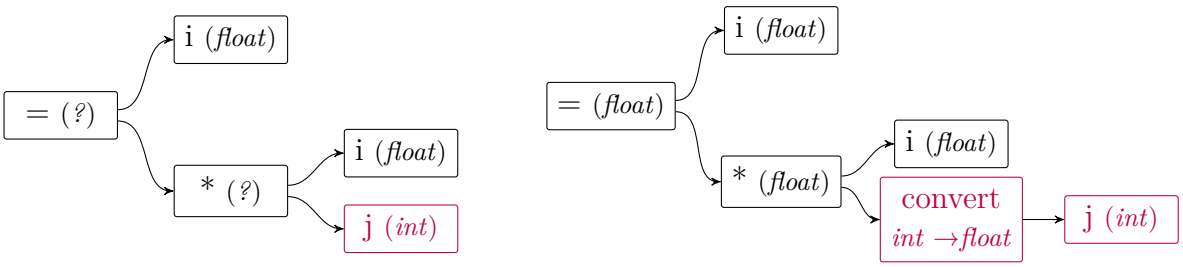

Abbildung 4: AST des Ausdrucks `i = i * j`, wobei `i` vom Typ `float` und `j` vom Typ `int` ist. Das Hinzufügen einer impliziten Typumwandlung ermöglicht so das Ermitteln des Ergebnistyps.

wird dann die eigentliche Binärdatei erzeugt. Bei LLVM heißt der Zwischencode ganz einfach *LLVM IR*, was für *LLVM intermediate representation* steht [Wik].

Die Vorteile von der Nutzung von Zwischencode liegen darin, dass dieser unabhängig von der Zielarchitektur ist. Man kann somit Optimierungen vornehmen, ohne dabei für jede Architektur einen eigenen Optimierer zu programmieren [Nov, S. 178]. Das Beispiel Java zeigt zudem, dass man Zwischencode auch dafür nutzen kann sogenannte *just in time* Kompilierung (kurt *JIT*) zu nutzen. Hierbei wird der eigentlich ausführbare Binärcode erst während der Ausführung erzeugt und kann auch während der Ausführung passend zum Laufzeitverhalten der Anwendung optimiert werden.

2.5.1 LLVM intermediate representation (LLVM IR)

Wie oben erwähnt, hat LLVM einen eigenen Zwischencode, der LLVM IR genannt wird [Wik]. Möchte man einen eigenen Compiler mittels LLVM entwickeln, so sollte man sich unbedingt den Zwischencode genauer anschauen, da LLVM die Elemente dieses Codes nahezu direkt als C++ Klassen abbildet.

Um näher drauf einzugehen ist in Listing 4 ein Beispiel eines einfachen C Programms zu sehen. Listing 5 zeigt den dazugehörigen LLVM Zwischencode, den man mit `clang` erzeugen kann[1].

```
1 int main() {
2     int a = 3;
3     int b = 5;
4     return a * b;
5 }
```

Listing 4: Beispiel C-Programm.

[1] `clang -O0 -S -emit-llvm -fno-discard-value-names example.c`

```
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 3, i32* %a, align 4
  store i32 5, i32* %b, align 4
  %0 = load i32, i32* %a, align 4
  %1 = load i32, i32* %b, align 4
  %mul = mul nsw i32 %0, %1
  ret i32 %mul
}
```

Listing 5: Beispiel LLVM IR zum C-Beispiel aus Listing 4

Was zunächst auffällt ist, dass der Zwischencode weniger nach C, sondern bereits sehr nach Assembler aussieht. Dennoch kann man die Parallelen zum originalen C Code erkennen: Es wird eine Funktion `@main` deklariert, welche einen 32 Bit Integer (`i32`) zurückgibt, und auch die Variablen `a` und `b` sind vorhanden.

Mit `%a = alloca i32` wird ein 32-Bit Integer alloziert und ist über die Variable `%a` erreichbar. Mit dem `store` Befehl wird der initiale Wert `3` in der Variable gespeichert.

Da die LLVM IR nur SSA Anweisungen kennt (man also keine Variablen mehrfach belegen kann), werden Rechenergebnisse in neuen Variablen gespeichert, wie es bei der `mul` Anweisung passiert.

Wie man auch erkennen kann, sind die Variablen `%0` und `%1` nicht notwendig. Der hier von `clang` erzeugte Zwischencode enthält keinerlei Optimierungen, diese werden jedoch mit der Nutzung verschiedener Compiler-Flags (zum Beispiel `-O2`) angewendet.

3 DIY Compiler

Im Rahmen dieser Seminararbeit habe ich einen vereinfachten Compiler geschrieben um die bis hierhin beschriebenen Konzepte zu verdeutlichen. Bedient habe ich mich des LLVM Frameworks, wodurch der Compiler selbst – wie LLVM auch – in C++ geschrieben ist.

3.1 Die Sprache

Die Sprache für diesen selbstgebauten Compiler sollte natürlich simpel und vor allem leicht zu parsen sein. Mein Ziel war es mit der Sprache Variablen zu deklarieren und Werte zurückgeben zu können.

Um das Parsen der Ausdrücke zu vereinfachten, ist alles in der Syntax klassischer Funktionsaufrufe gehalten. So beendet man beispielsweise mit `ret(42)` das Programm mit dem Exit-Code `42`. Auch gibt es für Variablen derzeit nur einen Typ: 32 bit große Integer.

Die komplette BNF der Sprache ist dementsprechend einfach und sieht wie folgt aus:

```
1 return-statement ::= return(<number>) | return(<identifier>)
2
3 declaration-statement ::= var(<identifier>)
4
5 identifier ::= {alpanumeric string}
6 number ::= {valid c++ integer value}
```

Ein Codebeispiel, welches ich weiter unten nochmal aufgreifen werde, deklariert eine Variable `i`, initialisiert diese direkt mit dem Wert `100` und gibt diese Variable als Exit-Code zurück.

```
1 var(i 100)
2 ret(i)
```

3.2 Lexer

Aus Gründen der Einfachheit, habe ich die Implementation des Parsers angepasst, dieser Fragt beim Lexer nach dem nächsten Token, anstatt diese passiv übergeben zu bekommen. Der Lexer bietet also eine Funktion `next_token(token_t* tok)` an, wodurch das nächste Token `tok` ermittelt wird.

Hier werden zunächst grundsätzlich Leerzeichen übersprungen und nur Buchstaben, Zahlen und runde Klammern gelesen. Bei gelesenen Wörtern, wird zudem Überprüft, ob das gelesene Wort ein Schlüsselwort (beispielsweise `var`) ist.

```
1 while (isalnum(peek_char())) {
2     identifier += pop_char();
3 }
4
```

```
5  if (identifier == "var") {
6      tok->tok_type = TOK_VAR;
7  else {
8      tok->tok_type = TOK_IDENT;
9  }
```

Listing 6: Alle alphanumerischen Zeichen werden gelesen und darauf überprüft, ob ein Schlüsselwort gelesen wird. Wenn dies nicht der Fall ist, handelt es sich im einen allgemeinen Identifier (zum Beispiel ein Variblenname).

3.3 Parser

Um auch beim Parser die Einfachheit des Compilers zu erhalten, wird nach erkannten Sequenzen von Token direkt ein Knoten im AST eingebaut und der dazugehörige Code erzeugt. LLVM speichert den erzeugten Code allerdings zunächst sogenannten LLVM-Context zwischen, sodass Optimierungen gegebenenfalls später noch angewendet werden können.

Das erkennen der Token und entsprechende Erzeugen der AST Knoten, geschieht in einer simplen Schleife. Dies ist das in Abschnitt 2.3.2 erwähnte „pattern matching“, bei dem versucht wird zu einer Tokenfolge einen passenden AST-Knoten zu finden.

```
1  void parse()
2  {
3      // initialize LLVM code generation
4
5      do {
6          token_t tok;
7          int ret = next_token(&tok);
8
9          if (ret == LEX_END) {
10             break;
11         }
12
13         switch (tok.tok_type) {
14         case TOK_VAR: {
15             std::unique_ptr<ASTVariableDeclarationExpression> expr
16                 = parse_variable_declaration();
17             expr->codegen();
18             break;
19         }
20     } while (true);
```

```
21
22      write_obj_file();
23  }
```

Listing 7: Vereinfachter Auszug aus der `parse()` Funktion mit Behandlung der Token für eine Variablendeklaration.

Die Funktion `parse_variable_declaration()` liest die nächsten Token ein und versucht den BNF Ausdruck `declaration-statement ::= var(<identifier>)` zu parsen. Führt dies zum Erfolg, wird der Variablenname in einer einfachen Symboltabelle gespeichert, welche lediglich eine Liste von Namen ist (`std::vector<std::string> symbol_table`).

Die Codeerzeugung nach dem Parsen geschieht ebenfalls über LLVM Klassen und einen `llvm::IRBuilder<>`, welcher dazu dient LLVM IR zu erzeugen, welches man wiederum beispielsweise in eine Datei schreiben kann. Bei der Codeerzeugung habe ich eine weitere Symboltabelle angelegt, um benutzte Variablen im LLVM IR zu referenzieren.

```
1  llvm::Type* type = llvm::Type::getInt32Ty(llvm_context);
2  llvm::AllocaInst* alloc_inst = llvm_builder.CreateAlloca(type, 0, name);
3
4  llvm_symbol_table[name] = alloc_inst;
```

Listing 8: Codeerzeugung einer einfachen Variablendeklaration vom Typ eines 32 bit Integers mit dem Startwert 0.

Das anschließende Erzeugen einer Objektdatei mittels `write_obj_file`, welche derzeit noch mit Hilfe von GCC in eine ausführbare Datei gelinkt werden muss, ist aufwändiger und da es nicht zwingend Teil des Compilers ist, möchte ich nicht ins Detail gehen. Benutzt habe ich im Wesentlichen ein Objekt der `llvm::TargetMachine` Klasse, dem man einen `llvm::legacy::PassManager` übergibt, welcher wiederum den Inhalt der Objektdatei in einen stream schreibt.

3.4 Beispiel und proof-of-concept

Als Beispiel sei folgendes, simples Programm gegeben. Es erstellt eine Variable `i` mit dem initialen Wert 100 und gibt diese zurück.

```
1  var(i 100)
2  ret(i)
```

Beim Kompilieren, werden zwei Daten geschrieben: Die Datei `out.ll` mit dem LLVM IR und die Objektdatei `out.o`.

```
define i32 @main() {
main:
        ; var(i 100)
        %i = alloca i32
        store i32 100, i32* %i

        ; ret(i)
        %i1 = load i32, i32* %i
        ret i32 %i1
}
```

Listing 9: Inhalt der `out.ll` Datei mit dem erzeugten LLVM Zwischencode.

```
# Kompilieren des Compilers
$> clang++ -c main.cpp

# Ausführen des Compilers. Es wird die Datei "test.lang" kompiliert
$> ./main
$> ls out*
out.ll  out.o

# Linken und ausführen des kompilierten Programms
$> gcc out.o -o out
$> ./out

# Ermitteln des Rückgabewertes
$> echo $?
100
```

4 Zusammenfassung

Nach einer kurzen Einführung in Compiler, deren Ergebnisse und einer Beschreibung vom Compiler-Framework LLVM, folgte direkt die Einleitung in den Aufbau eines Compilers. Dieser besteht aus den Phasen der lexikalischen, syntaktischen und semantischen Analyse gefolgt von der Code Erzeugung. Alle Phasen wurden dann anhand von Beispielen im Detail beleuchtet.

So wird bei der lexikalischen Analyse der Quelltext in eine Abfolge von Token umgewandelt, welche der syntaktischen Analyse übergeben wird. Diese erzeugt aus der Liste von Token eine Baumstruktur, den sogenannten abstrakten Syntaxbaum (kurz AST), welcher unter anderem die Schachtelung von Code abbilden kann. Aufgebaut wird der

AST anhand einer Syntaxbeschreibung, wozu man beispielsweise die Backus-Naur Form nutzen kann. Die semantische Analyse erhält den AST, fügt Typinformationen hinzu, passt den Baum an und übergibt in schließlich an die letzte Phase, der Code Erzeugung. In dieser letzten Phase wird der Baum in Zwischencode umgewandelt, welcher optimiert und am Ende in eine Objektdatei oder direkt eine ausführbare Datei kompiliert wird.

Mit diesem Wissen habe ich grob beschrieben, wie man mit verhältnismäßig wenig Aufwand einen eigenen Compiler mit Hilfe des LLVM Frameworks programmieren kann. Komplexere Sprachelemente (wie zum Beispiel `if` Anweisungen, Schleifen, Funktionen oder verschiedene Typen) kommen jedoch mit mehr Entwicklungsaufwand einher, wobei LLVM glücklicherweise den komplexen Teil der Codeerzeugung und Optimierung übernimmt.

Das Innenleben von Compilern ist durchaus komplex und nicht einfach zu entwickeln, Compiler folgen jedoch häufig gleichen oder zumindest ähnlichen Prinzipien. Die Entwicklung eines eigenen Compiler ist also durchaus möglich, wenn auch sehr Aufwändig.

Literatur

[Gup] Ajay Gupta. "The syntax of C in Backus-Naur Form". In: (). URL: https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm.

[Nov] Diego Novillo. *From Source to Binary: The Inner Workings of GCC*. URL: https://web.archive.org/web/20160410185222/https://www.redhat.com/magazine/002dec04/features/gcc/.

[Rog] Axel Rogat. "Aufbau und Arbeitsweise von Compilern". In: (). URL: http://www2.math.uni-wuppertal.de/~axel/skripte/compiler/comp.html.

[Wat17] Des Watson. *A Practical Approach to Compiler Construction*. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-52789-5.

[Wik] Wikipedia. *LLVM*. Letzter Zugriff: 12.04.2020. URL: https://de.wikipedia.org/wiki/LLVM.