Todo list

Ostfalia
Hochschule für angewandte Wissenschaften

Wolfenbüttel

Fakultät Informatik

Modul (AKRNM),
(W|S)S-JJJJ[/YYYY]

Projekt

Projekttitel oder -kurzbeschreibung

Autor :
Studierende oder Studierender, Mat.-Nr.: 00000000

Betreuer:
Dozentin oder Dozent
Betreuerin oder Betreuer

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Ich versichere weiterhin, alle wörtlich oder sinngemäß aus anderen Quellen übernommenen Aussagen als solche gekennzeichnet zu haben. Die eingereichte Arbeit ist weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen.

Ort, Datum	- Studierende oder Studierender -

Kapitel 0

Einleitung und Motivation

Beschreibung der Projektaufgabe

Benennung der Projektziele

Teil I

Compiler Theorie

Kapitel 1
Compiler Uebersicht

1.1 Todo Liste

- Was ist ein Compiler?
- Compiler Historie
- Einleitung zu den detaillierteren Beschreibungen:
  - Arten von Compilern
    -> Spaeter genauer erklaert: Top-Down Compiler
  - Compilerphasen
    * Einpass vs Mehrpass vs Zweipass zur Trennung Frontend <-> Backend
    -> Spaeter genauer erklaert: Zweipass Compiler fuer separate Backends zukuenftige Optimierungsschritte
  - Parser: Arten (EBNF etc)
    -> Spaeter genauer erklaert: EBNF
  - Zwischensprache: Vereinfacht Optimierungen, separierte Backends und Frontends
    -> Spaeter genauer erklaert: ??? (Wie so ein Zwischenstate aussehen soll ist scheinbar schwarze Magie, selber rausfinden)
  - Backends: Code Erzeugung etc
    -> Spaeter genauer erklaert: RISC Binary, RISC assembly
  - Bootstrapping compiler in eigener Sprache
    -> Passiert hier nicht, zu komplex und wild

1.2 Compilerhistorie

Die ersten Computer wurden direkt in Maschinensprache programmiert. Mithilfe von handgeschriebenen Notationen und Tabellen haben die Programmierer die Übersicht über die Programme behalten.
Mit der Zeit wurden die Aufgaben, die die Maschinen Computer erledigen sollten, immer komplexer und größer. Über die binären Instruktionen den Überblick zu behalten, wurde damit auch schwieriger. Statt direkt in Binärcode zu arbeiten, war es leichter, die Operationen in Symbole wie “LOAD” und “ADD” zu schreiben und zu lesen. Diese Symbole wurden dann in einem

Zwischenschritt durch einen symbolbasierten Assembler in die computer-verständliche Maschinensprache übersetzt.
Zusätzlich dazu veränderte sich die Technik; Mit der Einführung der Magnettrommelspeicher zum Beispiel kamen Herausforderungen in der Optimierung des Speicherzugriffs auf. Je nach Ort der Daten konnte die Dauer, auf die Daten mit Direktzugriff zuzugreifen, um einen Faktor von 50 variieren [2]. Mithilfe eines optimierenden Assemblers wurden die Adresszuweisungen der Daten optimiert, sodass die Zugriffsdauer auf diese Daten verringert wurde.
Die Instruktionstypen, die eine damalige Prozessorarchitektur der Computer lesen und bearbeiten konnte, waren sehr limitiert. Oftmals waren mehrere kleine Instruktionen nötig, um Schritte im Programm darzustellen, die mit einem größeren, expliziteren Instruktionstypen zusammengefasst werden könnten. Diese fehlenden Instruktionstypen wurden anfangs mit einem Interpreter hinzugefügt; Ein Programm, das die Übersetzung der programmierten, komplexeren Befehle in für die Prozessorarchitektur lesbare Befehle übernahm. Da ein Interpreter selbst auf dem Computer ausgeführt werden muss, führt dies zu deutlichem Mehraufwand in der Ausführung der Programme. Die ersten Compiler wurden geschrieben, um dieses Problem zu lösen. Sie übersetzten die vom Programmierer geschriebenen Befehlslisten in für die Prozessorarchitektur des Computers verständliche Instruktionen. Die Compiler übernahmen mit der Zeit weitere Aufgaben wie Speicherzugriffsoptimierungen und dynamische Speicherallokation. Verschiedene Programmiersprachen abstrahieren die Prozessorarchitektur weiter. Sie vereinfachten die Implementierung von Programmen in domänenspezifischen Problemen wie Handel und Forschung und verbreiteten sich später mit dem Aufkommen von Allzweck-Programmiersprachen auch darüber hinaus[2].

1.3 Aufbau eines Compilers

Heutzutage gibt es eine große Bandbreite an Programmen, die Compiler genannt werden. Der Compiler im klassischen Sinne übersetzt in Programmiersprachen geschriebenen Programmcode in Maschinencode. Die “Cross-Compiler” hingegen übersetzen von einer Programmiersprache in eine andere Programmiersprache. Zusätzlich dazu gibt es Compiler (oder auch “compilerähnliche Werkzeuge”) die auf ganz anderen Datenstrukturen arbeiten und zum Beispiel Logdateien in Tabellen kompilieren[4]. Im Rahmen dieser Arbeit wird der klassische Compiler behandelt, mit dem eine Programmiersprache zu einer Maschinensprache kompiliert wird.
Diese Compiler haben generell 6 Phasen, in denen sie arbeiten:
Die *Lexikalische Analyse* unterteilt das Programm in Tokens, die Einheiten wie Variablennamen oder Operationszeichen aufteilen.
Diese Tokens werden dann an die Syntaxanalyse geschickt, wo mithilfe grammatikalischer Regeln die Struktur des Programmcodes überprüft wird. Die Definition dieser Regeln zusammen mit dem Interpretieren der Tokens wird Parser genannt. Der Parser ordnet die einzelnen Tokens mitsamt der Regeln in eine Baumstruktur ein. Es gibt zwei grundlegende Arten von Parsern im Compilerbau; Der Top-Down Parser baut den Syntax Baum aus der Wurzel heraus und bricht dann die Tokens immer mehr im Detail runter, während der Bottom-Up Parser mit den einzelnen Tokens anfängt und diese zusammenfasst um zu einem Syntax Baum zu kommen[3].

Bild Top Down vs Bottom Up Tree

Nach dem Parsen wird eine semantische Analyse auf dem Syntaxbaum ausgeführt, etc etc etc

Kapitel 2

Der Top-Down Compiler

Uebersicht und Beschreibung Top-Down Compiler, Verwendung und Software Stacks
Vorteile / Unterschiede in Implementierung gegenüber Bottom-Up
Mit EBNF Parser als Kapitel kombinieren?
Schwierigkeiten beim lösen der Grammatiken + Backtracking aus [1, S. 46-51]

Kapitel 3

Der Compiler im Zweipass Verfahren

Aufbau und Details zum Zweipass Verfahren

Kapitel 4

Der EBNF-basierte Parser

Historie zu EBNF, Verwendungsbeispiele, etc

Kapitel 5

Die Zwischensprache im Compiler mit Zweipass Verfahren

Ziele der Zwischensprache, Aufbau, Verwendung, Aufzeigen moeglicher Optimierungen

Vielleicht zu wenig Referenz Material fuer Theorie vor der Umsetzung, stattdessen eher die Umsetzung im Fazit Teil der Arbeit dokumentieren?

Kapitel 6

Das RISC Binaercode erzeugende Backend

- Aufgabe des Backends
- Beschreibung Zielplattform RISC
  - Detaillierte Uebersicht ueber Assembly, Call Stack und weitere Themen

Teil II

Umsetzung des Compilers

Kapitel 7

Planung Des Compilers

Die Implementierung des Compilers soll folgende Ziele erfuellen:

- Verstaendlichkeit und Inspizierbarkeit: Der Compiler soll inspizierbar und verstaendlich aufgebaut sein. Zusaetzlich soll die Arbeitsweise des Compilers transparent sein, indem die einzelnen Teile des Compilers gut debugbar oder inspizierbar sind.
- Flexibles Backend: Der Compiler soll in der Lage sein, auf verschiedene Zielplattformen wie RISC Assembly und RISC Binary zu kompilieren.

Weitere Aufgaben:

- Die generierte RISC Binary kann irgendwie ausgefuehrt werden

Kapitel 8

Umsetzung des Parsers

Wie sieht die Sprache aus, wie parst der diese Sprache, wie mappe ich das auf die Zwischensprache.

Weitere Details: Speichereffizienz, Geschwindigkeit des Parsers, EBNF trickery (zur Optimierung von Geschwindigkeit etc), Fehlerbehandlung der Input Datei (Syntax Fehler / Logik Fehler / etc)

Kapitel 9

Umsetzung der Zwischensprache

Aufbau, wie halte ich Frontends und Backends kompatibel ohne zu sehr auf eine Architektur zu schielen mit der Datenstruktur

Weitere Details: Speichereffizienz, Geschwindigkeit, Fehlerbehandlung

Kapitel 10

Umsetzung des RISC Backends

Wie mappe ich die Zwischensprache auf den Output, wieviel Code kann ich zwischen RISC binary und RISC Assembly sharen

Weitere Details: Speichereffizienz, Geschwindigkeit, Fehlerbehandlung

Kapitel 11

Umsetzung des RISC Interpreters

REDACTED: Gibts bereits,ist nicht in Aufgabenstellung, z.B.: https://www.cs.cornell.edu/courses/cs3410/2
(=> dafuer sorgen das RISC Assembly mit z.B. diesem Interpreter [besser irgendwas open source maessiges] kompatibel ist)

Teil III

Auswertungen und Fazit

Kapitel 12

Inspektion des Compilers

Performance des Compilers, Bugs?, Debuggability

Kapitel 13

Vergleich mit anderen, echten Compilern

Vergleich von Output von Beispiel Code, der gleiches tut, ueber mehrere andere Programmiersprachen und ihre Compiler. Analyse, Ausblick auf Optimierungsmoeglichkeiten.

Vielleicht witzig: Gibt bestimmt einen Compiler Compiler mit RISC Backend, in den ich ein bisl EBNF zum parsen "meiner"Prog Sprache packen kann -> direkter Vergleich zum handgeschriebenen compiler

Kapitel 14

Vorlage

Einleitung

14.1 Sektion

Text

14.1.1 Subsektion

Text

14.2 Fazit

Text

Kapitel 15

Schluss und Fazit

Schluss und Fazit

Literaturverzeichnis

[1] W.A. Barrett. *Compiler Construction: Theory and Practice.* Computer Science Series. Science Research Associates, 1986.

[2] Peter Calingaert. *Assemblers, Compilers, and Program Translation.* W. H. Freeman & Co., USA, 1979.

[3] A. Meduna. *Elements of Compiler Design.* CRC Press, 2007.

[4] H. Mössenböck. *Compilerbau: Grundlagen Und Anwendungen.* Lehrbuch. dpunkt.verlag, 2024.

Anhang A

Anhang A

siehe nächste Seite

Der Anhang enthält auf den nächsten Seiten [...].

Anhang B

Anhang B

Anhang C