

**Dokumentation:**

**Systemnahes Programmieren**

Konstantin Krause

Noah Percifull

Philipp Lehmann

Matthäus Prasse

**WS 2019/2020**

Inhaltsverzeichnis

[1 Einleitung 3](#_Toc17642446)

[2 Scanner 4](#_Toc17642447)

[2.1 Aufgabenstellung Scanner 4](#_Toc17642448)

[3 Parser 5](#_Toc17642449)

[3.1 Aufgabenstellung Parser 5](#_Toc17642450)

[Literaturverzeichnis 6](#_Toc17642451)

[Abbildungsverzeichnis 7](#_Toc17642452)

[Tabellenverzeichnis 8](#_Toc17642453)

[Abkürzungsverzeichnis 9](#_Toc17642454)

# Einleitung

Das Labor Systemnahes Programmieren dient dazu, den Studierenden einen tieferen Einblick in die Funktionsweise eines Compilers zu gewähren und zusätzlich ihre Kenntnisse in C/C++ zu verbessern. Hauptaufgabe eines Compilers ist es, Quellcode einer Programmiersprache in eine für den Computer lesbare Form zu übersetzen.

Das Labor ist in die Teilaufgaben Scanner und Parser unterteilt.

# Scanner

## Aufgabenstellung Scanner

Die erste Teilaufgabe beschäftigt sich mit der Lexikalischen Analyse, also der Zerlegung des Quellcodes in seine Bestandteile und der Erzeugung der entsprechenden Zwischendarstellung.

## Architektur

Die Implementierung ist in verschiedene Klasse unterteilt.

### Buffer

### Scanner

### SymTable

In der SymTable werden Referenzen auf die StringTable gespeichert. Damit ist es möglich per Hash auf die einzelnen Strings zuzugreifen. Der interne struct Key beschreibt hierbei die genaue Speicherstelle, was einen schnellen Vergleich von Lexemen ermöglicht.

### StringTable

Die StringTable speichert alle Lexeme. Dieser Speicher ist als Array realisiert. Wenn die maximale Größe es Array erreicht ist, wird ein neues Array mit doppelter Größe erstellt, welches den Inhalt des alten Arrays übernimmt.

### Automat

Der Automat überprüft die verschiedenen, vom Scanner zur Verfügung gestellten Symbole. Die eingelesenen Symbole werden in einer Verketteten Liste zwischengespeichert. Aus den Symbolen baut der Automat die verschiedenen Tokens. Um die jeweils korrekten Token erzeugen zu können, durchläuft der Automat beim Einlesen der Symbole verschiedene Zustände.

### LinkedList

Diese Klasse implementiert eine Verkettete Liste. In ihr werden eingelesene Symbole zwischengespeichert, wobei jedes Symbol entweder am Anfang oder am Ende der Liste eingefügt werden können. Es kann nur das Symbol am Kopf der Liste ausgelesen werden, wodurch das Symbol aus der Liste entfernt wird. Die Länge der Liste ist nicht begrenzt.

### Information

Über Information erfolgt der Zugriff in die StringTable auf die gespeicherten Lexeme.

### InformationLinkedList

Die InformationLinkedList wird benutzt für den Fall, dass zwei Mal derselbe Hash erstellt wird. In diesem Fall werden in der SymTable die zwei Informationen aneinander gekettet, was den Zugriff auf die verschiedenen Lexeme erlaubt, obwohl sie den selben Hash besitzen.

### Tokenlist

# Parser

## Aufgabenstellung Parser

In Aufgabenteil 2 sollen die in Aufgabenteil 1 erstellte Sequenz von Tokens in funktionierenden Maschinencode umgewandelt werden. Dafür müssen die Tokens erstmal in eine geeignete Struktur gebracht werden. Dies erfolgt über den ParseTree. Die Grammatik des Codes wird bei der Überführung in den Baum überprüft. Anschließend wird aus den Baumknoten der Maschinencode erzeugt.

## Änderung des Automaten

Zusätzliche Tokentypes wurden hinzugefügt, um die Funktion READ und WRITE umsetzen zu können.

## ASTCreator

Der Abstract Syntax Tree erstellt die Baumstruktur. Um das zu gewährleisten besitzt die Klasse einen internen Automaten, der die Grammatik überprüft. Dies wird über verschiedene Methoden und Switch-Cases durchgeführt. Wenn ein Fehler erkannt wird, terminiert der Automat frühzeitig um Arbeitszeit zu sparen.

## ASTStack

Der ASTStack wird intern vom ASTCreators verwendet. Er verhindert, dass beim Bauen eines Programms es durch zu viele rekursive Aufrufe es zu einem Stack Overflow kommen kann. Noch zu verarbeiten Tokens werden zwischengespeichert um so dem ASTCreator mitzuteilen, welche Nodes als nächstes erstellt werden muss.

## ASTNode

Der ASTNode beschreibt die einzelnen Knoten des Baums. Die Navigation erfolgt über die Referenzen in den Eltern- und Kinderknoten. Hierbei gibt es drei Arten von Knoten, die Nicht-Terminalen, welche die Struktur beschreiben, die Terminalen welche Programmabschnitte beschreiben, die nicht weiter expandiert werden können, wie zb, Rechenzeichen und die Knoten welche Wertigkeiten beschreiben. Dies können Zahlen sein, die direkt gespeichert werden oder Referenzen auf selbsterstelle Variablen.

# Literaturverzeichnis

**Im aktuellen Dokument sind keine Quellen vorhanden.**

# Abbildungsverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

# Tabellenverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

# Abkürzungsverzeichnis

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |