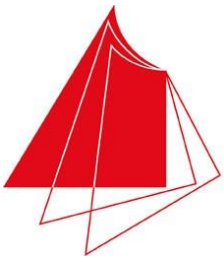


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Dokumentation
Systemnahes Programmieren

Konstantin Krause
Noah Percifull
Philipp Lehmann
Matthäus Prasse

SS 2019

Inhaltsverzeichnis

1 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, den Maschinencode, zu übersetzen.

Das Labor ist in die Teilaufgaben Scanner und Parser unterteilt. In der lexikalischen Analyse des Scanners wird der Quellcode entsprechend der vorgegebenen Grammatik zerlegt. Die syntaktische Korrektheit wird während der Erstellung der Parse-Baums durchgeführt und dessen Struktur für die Erzeugung des Maschinencodes weiterverwendet.

2 Scanner

2.1 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 in Form von Tokens.

2.2 Architektur

Die Implementierung ist in verschiedene Klasse unterteilt. Hierbei stellt die Klasse Scanner die Schnittstelle dar, auf welcher der Parser später aufbaut.

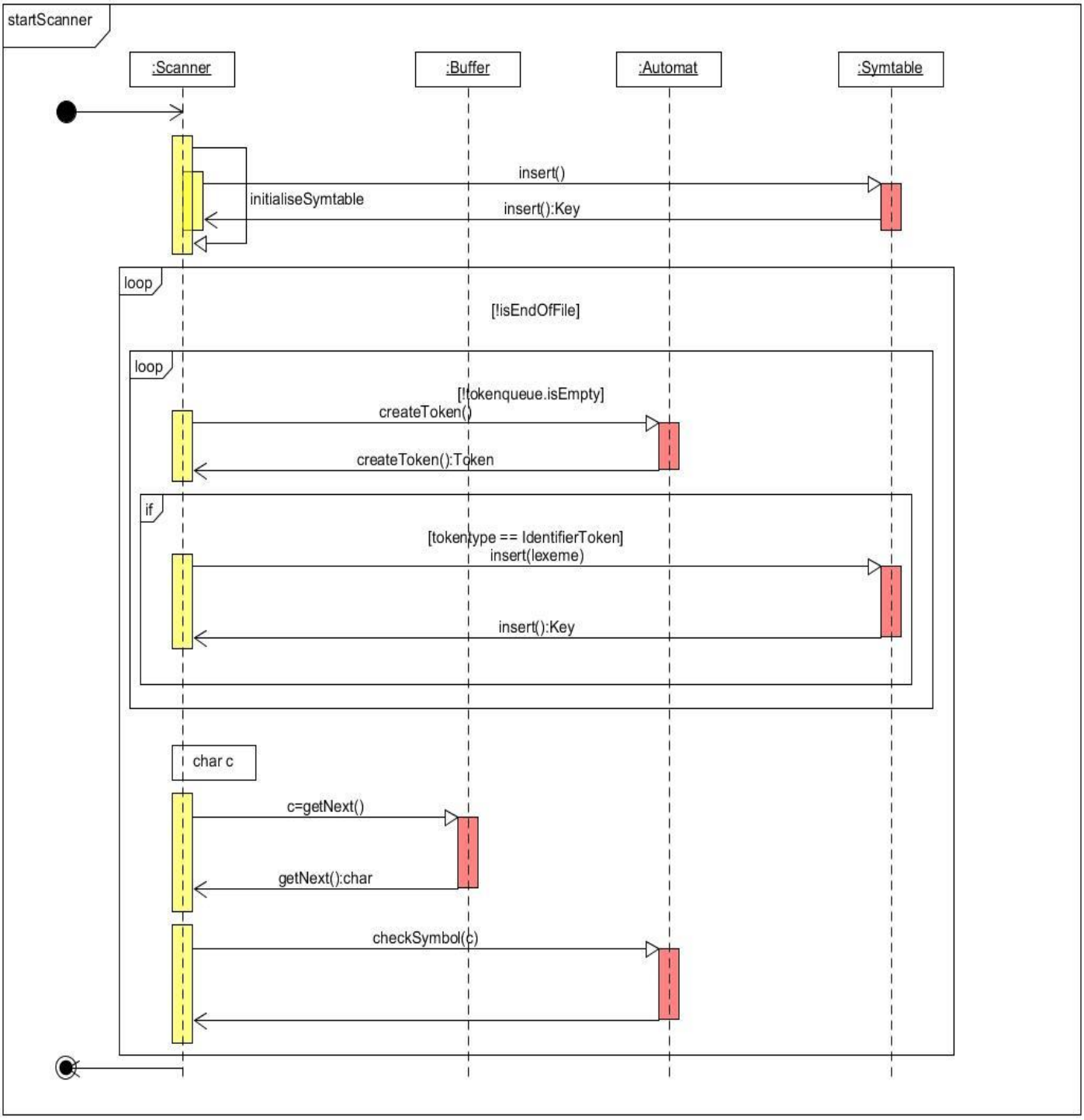
2.2.1 Buffer

Der Buffer hat die Aufgabe, die Quellcode Datei einzulesen und diese zeichenweise zur Verfügung zu stellen. Da diese nicht in ihrer Gesamtheit zwischengespeichert wird, muss sich gemerkt werden, wie viel der Datei bereits verwertet wurde. Hierbei ist der Buffer in zwei Char-Pointer Partitionen unterteilt, die abwechselnd befüllt werden, nämlich genau dann, wenn die aktuell verwendete Hälfte vollständig ausgelesen wurde. Dieses Konzept ist an den sogenannten Ringbuffer angelehnt, der die am längsten nicht verwendeten Zeichen fortlaufend beim Auslesen überschreibt.
Dies ermöglicht prinzipiell die „unget“ Funktionalität, welche hier allerdings durch die Architektur des Automaten nicht notwendig ist.

2.2.2 Scanner

Die Klasse Scanner stellt die Schnittstelle nach Außen dar und ist damit auch die verwaltende Klasse des Teilprojekts. Sie erstellt den Buffer, initialisiert die Symboltabelle mit allen validen Zeichenkombinationen, welche weder Schlüsselwort, Lexem oder Zahl sind, und kontrolliert den Zugriff auf den Automaten. Es wird außerdem das Fertigstellen der Identifier- und Sign-Tokens bezüglich des Eintrags in die Symboltabelle und deren Verknüpfung des von dieser zurückgelieferten Schlüssels durchgeführt.
Nach Außen stehen folgende Methoden zur Verfügung:

- startScanner: Tokens werden mit Hilfe des Automaten erzeugt und so lange Zeichen aus dem Buffer ausgelesen und an den Automaten weitergereicht, bis die Eingabedatei vollständig ausgelesen wurde
- nextToken: Liefert den obersten Token in der Token Queue
- hasTokens: Information über den Zustand der Token Queue
- isWithoutErrors: Information, ob während dem Ablauf der Funktion startScanner Fehler innerhalb der Eingabedatei bemerkt wurden



1 System-Sequenz Diagramm (Scanner.startScanner)

2.2.3 Symtable

In der Symboltabelle 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, da man nur den Key vergleichen muss und nicht den Key. Für den Fall, dass zwei Lexeme denselben Hash haben werden die Informationen in einer der verketteten Listen der Symboltabelle gespeichert, um Fehler zu vermeiden. Beim Eintragen in die Symboltabelle wird geprüft, ob das Lexem schon eingefügt wurde. Hierfür wird eine Belegungsliste gepflegt, um die Performance bei einer hohen Anzahl an Lexemen zu gewährleisten. Ist dies der Fall wird der Key des identischen Lexems zurückgegeben. So können identische Lexeme erkannt werden.

2.2.4 StringTable

Die StringTable speichert alle Lexeme. Dieser Speicher ist als Array realisiert. Die Lexeme werden hier in Sequenz hintereinander eingetragen. die einzelnen Lexeme werden durch ein Trennzeichen voneinander getrennt.
Wenn die maximale Größe es Array erreicht ist, wird ein neues Array mit doppelter Größe erstellt, welches den Inhalt des alten Arrays übernimmt. Dies erlaubt das Speichern von beliebig vielen und beliebig großen Lexemen.

2.2.5 Automat

Der Automat überprüft die verschiedenen, vom Scanner zur Verfügung gestellten Symbole. Die eingelesenen Zeichen werden in einer Verketteten Liste zwischengespeichert. Für die Fehlerausgabe essenzieller Bestandteil ist das Nachverfolgen der Zeilen- und Spalten-Information, welche der Automat für die erkannten Symbole bereitstellt. Aus den Zeichen bereitet der Automat die verschiedenen Tokens vor. Um den jeweils korrekten Token erzeugen zu können, durchläuft der Automat beim Einlesen der Symbole verschiedene Zustände. Sobald ein gültiges Zeichen gelesen wird, dass gemäß den Bildungsregeln nicht mehr zum bisher gebauten Symbol gehören kann, wird das entsprechende Token vorbereitet und das neue Zeichen erneut durch den Check gegeben, um dieses dem nachfolgenden Token zuzuordnen. Die Token-Queue signalisiert dem Scanner, dass Tokens vorliegen, die vor dem nächsten Durchlauf des Automaten zu erzeugen sind und welchen Token Type diese haben.

2.2.6 LinkedList (deprecated)

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.

2.2.7 Information

Die Klasse Information stellt eine Wrapper Klasse dar, worüber der Zugriff in die StringTable auf die gespeicherten Lexeme erfolgt. Sie beschreiben die Speicherstelle der Lexeme in der String Table und geben eine direkte Referenz zu den Lexemen. Diese direkte Referenz erlaubt das direkte Vergleichen von zwei Lexemen, um zu bestimmen, ob diese identisch sind.
Information sollte außerdem im weiteren Projektverlauf durch den Parser für die Speicherung der Typ-Information der jeweiligen Nodes erweitert werden.

2.2.8 InformationLinkedList (deprecated)

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 denselben Hash besitzen.

3 Parser

3.1 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.

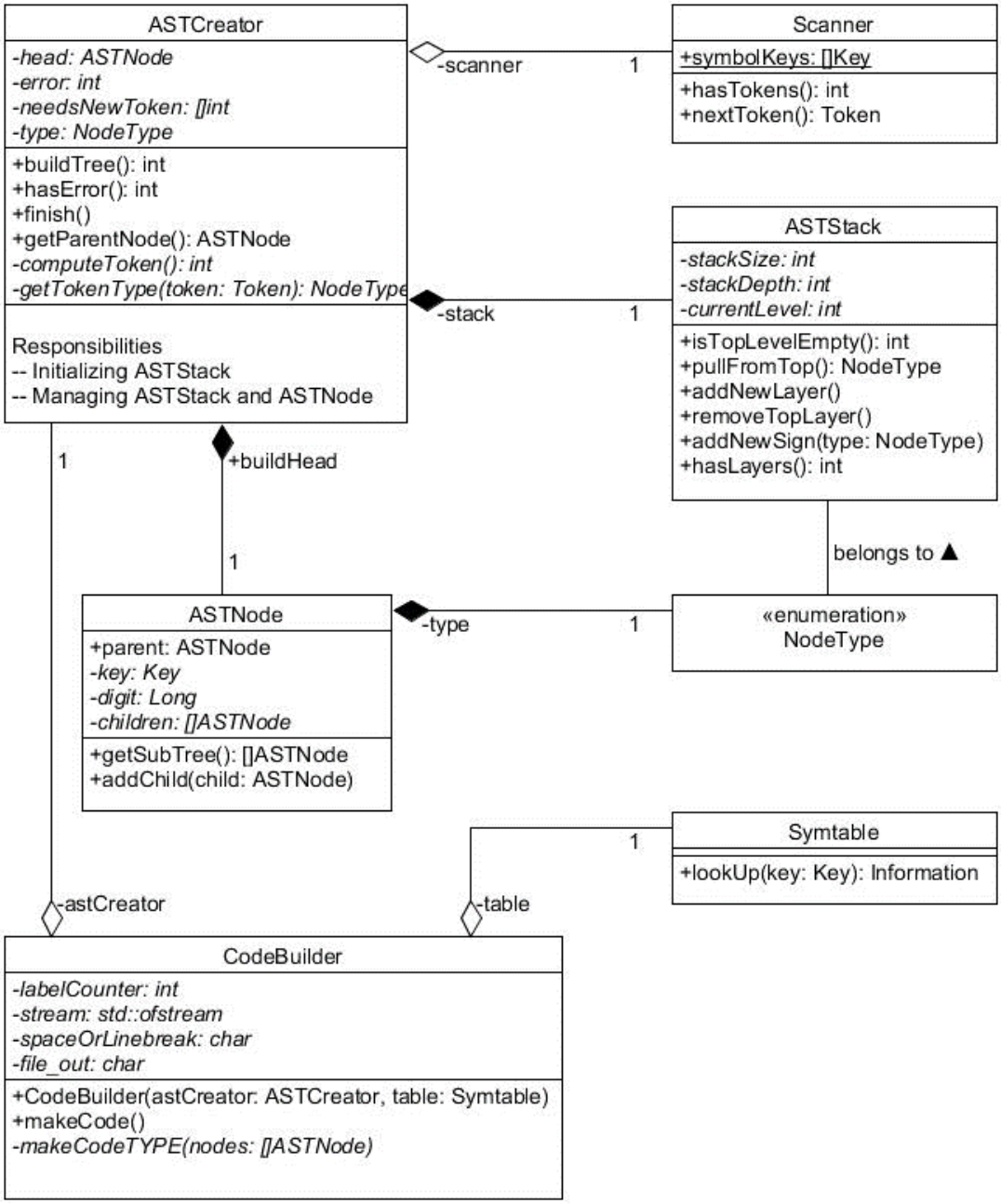
3.2 Änderung des Automaten

Zusätzliche Tokentypes wurden hinzugefügt, um die Funktionalität für die Schlüsselwörter „read“, „write“ und „int“ umsetzen zu können.

3.3 Generische doppelt verkettete Liste

Die Klasse *Link_List* ersetzt die bisher verwendeten typspezifischen Listen LinkedList, InformationLinkedList und TokenQueue. Diese ist generisch und beinhaltet doppelte statt einfach verkette Elemente, sowie die Möglichkeit Indexzugriffe durchzuführen.
Hierdurch konnten einige Fehlerquellen eliminiert und diverse Verbesserungen am Scanner durchgeführt werden

3.4 Architektur



2 Klassendiagramm (Parser)

3.4.1 ASTCreator

Der ASTCreator erstellt die Baumstruktur. Um das zu gewährleisten besitzt die Klasse einen internen Automaten, der die Grammatik überprüft. Zeichen werden hier eingelesen und entsprechende Zeichen, die zur grammatikalisch korrekten Abarbeitung benötigt werden, in einem Stack abgespeichert. Die Tokens werden

dann an den ASTCreator gegeben und der ASTCreator überprüft, ob die Zeichen der Tokens mit denen im Stack übereinstimmen.
Das Abgleichen der Zeichen wird in einem Switch-Case Block durchgeführt. Sollte ein Nicht-Terminal aus dem Stack verarbeitet werden bedeutet dies, dass weitere Zeichen auf den Stack gelegt werden. Zum Schluss überprüft der AST-Creator, ob er sich in einem passenden Terminalen Zustand befindet oder es noch Zeichen im Stack gibt. Wenn der Stack noch nicht vollständig gelehrt wurde wird geprüft, ob die übrigen Zeichen noch aufgelöst werden können.
Wenn der ASTCreator einen Fehler erkennt setzt er frühzeitig ein Fehler-Flag damit der Vorgang zeitnah abgebrochen werden kann.

3.4.2 ASTStack

Der ASTStack wird intern vom ASTCreator 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 müssen.
Der Stack hat mehrere Layer, welche die Hierarchie der Baumstruktur darstellt. Diese muss erhalten bleiben, da zu einem späteren Zeitpunkt ermittelt werden muss, welche Nodes mit welchen anderen verbunden sind.

3.4.3 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 z.B. Rechenzeichen und die Knoten welche Wertigkeiten beschreiben. Dies können Zahlen sein, die direkt gespeichert werden oder Referenzen auf selbsterstelle Variablen.
Die ASTNodes werden von dem CodeBuilder genutzt, um den Maschinencode zu erstellen. Die Bearbeitungsreihenfolge wird dabei durch die Hierarchische Struktur der Nodes beschrieben.

3.4.4 CodeBuilder

Die Klasse CodeBuilder erhält den ASTCreator als Schnittstelle und nutzt diese, um den erzeugten Parse-Baums zu traversieren. Hierbei werden die jeweiligen Kinder der Nodes genutzt, um gemäß den Regeln der zugrundeliegenden Grammatik abzusteigen; hierbei wird allerding explizit *nicht* die Syntax geprüft und sich auf das Vorhandensein der jeweils korrekten NodeType verlassen. Alle

möglichen, aber nicht zwingendermaßen vorhandenen Verschachtelungen werden durch rekursive Aufrufe berücksichtigt.
Die Implementation der Klasse ist insgesamt darauf ausgelegt, mit viel Redundanz seitens des Parse-Baums umgehen zu können und ignoriert so zum Beispiel sämtliche Klammern bzw. grundsätzlich für die Logik des Maschinencodes nicht relevanten Symbolen. Dies hat den Nachteil, dass Syntax-Fehler im Parse-Baum das Programm terminieren würde, allerdings sollten diese an erster Stelle durch die Analysearbeit des ASTCreator sowieso nicht auftreten. Entsprechend werden hier auch keine Fehler behandelt.

Abbildungsverzeichnis

*Alle verwendeten Abbildungen sind in eigenständiger Bearbeitung entstanden.