
ALBERT LUDWIGS UNIVERSITÄT FREIBURG

TECHNISCHE FAKULTÄT

PicoC-Compiler

Übersetzung einer Untermenge von C in den Befehlssatz der RETI-CPU

BACHELORARBEIT

Abgabedatum: 28th April 2022

Author:
Jürgen Mattheis

Gutachter:
Prof. Dr. Scholl

Betreuung:
M.Sc. Seufert

Eine Bachelorarbeit am Lehrstuhl für
Betriebssysteme

ERKLÄRUNG

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Inhaltsverzeichnis

1	Motivation	7
1.1	PicoC und RETI	7
1.2	Problemstellung	7
1.3	Eigenheiten der Sprache C	7
1.4	Richtlinien	7
2	Einführung	8
2.1	Compiler und Interpreter	8
2.1.1	T-Diagramme	8
2.2	Grammatiken	8
2.3	Grundlagen	8
2.3.1	Mehrdeutige Grammatiken	9
2.3.2	Präzidenz und Assoziativität	9
2.4	Lexikalische Analyse	9
2.5	Syntaktische Analyse	11
2.6	Code Generierung	12
2.7	Fehlermeldungen	12
3	Implementierung	13
3.1	PicoC und RETI	13
3.2	Grammatiken	13
3.2.1	Umstzung von Präzidenz	13
3.3	Lexikalische Analyse	14
3.3.1	Verwendung von Lark	14
3.3.2	Basic Parser	14
3.4	Syntaktische Analyse	14
3.4.1	Verwendung von Lark	14
3.4.2	Derivation Tree	14
3.4.3	Derivation Tree Vereinfachung	14
3.4.4	Early Parser	14
3.5	Code Generierung	14
3.5.1	Passes	14
3.5.2	Umsetzung von Pointern und Arrays	14
3.5.3	Umsetzung von Structs	14
3.5.4	Umsetzung von Funktionen	14
3.5.5	Umsetzung kleinerer Details	14
3.6	Fehlermeldungen	14
3.6.1	Error Handler	14
4	Ergebnisse und Ausblick	15
4.1	Funktionsumfang	15
4.2	Qualitätskontrolle	15
4.3	Kommentierter Kompilervorgang	15
4.4	Erweiterungsideen	15
A	Appendix	16
A.1	Konkrete und Abstrakte Syntax	16

A.2	Bedienungsanleitungen	16
A.2.1	PicoC-Compiler	16
A.2.2	Showmode	16
A.2.3	Entwicklertools	16

Abbildungsverzeichnis

Tabellenverzeichnis

3.1 Präzidenzregeln von PicoC	13
---	----

Definitionen

2.1	Compiler	8
2.2	Interpreter	8
2.3	T-Diagram	8
2.4	Sprache	8
2.5	Chromsky Hierarchie	8
2.6	Grammatik	8
2.7	Reguläre Sprachen	8
2.8	Kontextfreie Sprachen	9
2.9	Ableitungsbaum	9
2.10	Mehrdeutige Grammatik	9
2.11	Assoziativität	9
2.12	Präzidenz	9
2.13	Pattern	9
2.14	Lexeme	9
2.15	Lexer (bzw. Scanner)	10
2.16	Literal	10
2.17	Parser	11
2.18	Recognizer	11
2.19	Konkrete Syntax	11
2.20	Derivation Tree	11
2.21	Abstrakte Syntax	11
2.22	Abstrakte Syntax Tree	11
2.23	Transformer	11
2.24	Visitor	12
2.25	Pass	12
2.26	Fehlermeldung	12
3.1	Symboltabelle	14

1 Motivation

1.1 PicoC und RETI

1.2 Problemstellung

1.3 Eigenheiten der Sprache C

1.4 Richtlinien

2 Einführung

2.1 Compiler und Interpreter

Definition 2.1: Compiler

Definition 2.2: Interpreter

2.1.1 T-Diagramme

Definition 2.3: T-Diagram

2.2 Grammatiken

2.3 Grundlagen

Definition 2.4: Sprache

Definition 2.5: Chomsky Hierarchie

Definition 2.6: Grammatik

Definition 2.7: Reguläre Sprachen

Definition 2.8: Kontextfreie Sprachen**2.3.1 Mehrdeutige Grammatiken****Definition 2.9: Ableitungsbaum****Definition 2.10: Mehrdeutige Grammatik****2.3.2 Präzidenz und Assoziativität****Definition 2.11: Assoziativität****Definition 2.12: Präzidenz****2.4 Lexikalische Analyse**

Die **Lexikalische Analyse** bildet üblicherweise die erste Ebene innerhalb der **Pipe Architektur** bei der Implementierung von Compilern. Die Aufgabe der lexikalischen Analyse ist vereinfacht gesagt, in einem Inputstring, z.B. dem Inhalt einer Datei, welche in **UTF-8** codiert ist, Folgen endlicher Symbole (auch **Wörter** genannt) zu finden, die bestimmte **Pattern** (Definition 2.13) matchen, die durch eine **reguläre Grammatik** spezifiziert sind.

Definition 2.13: Pattern

Beschreibung aller möglichen **Lexeme** einer Menge \mathbb{P}_T , die einem bestimmten **Token** T zugeordnet werden. Die Menge \mathbb{P}_T ist eine möglicherweise unendliche Menge von **Wörtern**, die sich mit den Regeln einer **regulären Grammatik** G_{Lex} einer **regulären Sprache** L_{Lex} beschreiben lassen^a, die für die Beschreibung eines **Tokens** T zuständig sind.^b

^aAls Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.

^bWhat is the difference between a token and a lexeme?

Diese Folgen endlicher Symbole werden auch **Lexeme** (Definition 2.14) genannt.

Definition 2.14: Lexeme

Ein **Lexeme** ist ein **Wort** aus dem Inputstring, welches das **Pattern** für eines der **Token** T einer **Sprache** L_{Lex} matched.^a

^aWhat is the difference between a token and a lexeme?

Diese **Lexeme** werden vom **Lexer** im **Inputstring** identifiziert und **Tokens** T zugeordnet (Definition 2.15). Die **Tokens** sind es, die letztendlich an die **Syntaktische Analyse** weitergegeben werden.

Definition 2.15: Lexer (bzw. Scanner)

Ein **Lexer** ist eine *partielle* Funktion $lex : \Sigma^* \rightarrow (N \times W)^*$, welche ein **Wort** aus Σ^* auf ein **Token** T mit einem **Tokennamen** N und einem **Tokenwert** W abbildet, falls diese Folge von Symbolen sich unter der **regulären Grammatik** G_{Lex} , der **regulären Sprache** L_{Lex} ableiten lässt.^a

^alecture-notes-2021.

Der Grund warum nicht einfach nur die **Lexeme** an die **Syntaktische Analyse** weitergegeben werden und der Grund für die Aufteilung des **Tokens** in **Tokenname** und **Tokenwert** ist, weil z.B. die Bezeichner von Variablen, Konstanten und Funktionen beliebige Zeichenfolgen sein können, wie `my_fun`, `my_var` oder `my_const` und es auch viele verschiedenen Zahlen gibt, wie 42, 314 oder 12. Die Überbegriffe bzw. Tokennamen für beliebige Bezeichner von Variablen, Konstanten und Funktionen und beliebige Zahlen sind aber trotz allem z.B. **Zahl** und **Bezeichner**.

Ein **Lexeme** ist damit aber nicht das gleiche, wie der **Tokenwert**, denn z.B. im Falle von PicoC kann z.B. der Wert 99 durch zwei verschiedene Literale dargestellt werden, einmal als ASCII-Zeichen 'c' und des Weiteren auch in Dezimalschreibweise als 99¹. Der **Tokenwert** ist jedoch der letztendliche Wert an sich, unabhängig von der Darstellungsform.

Um Verwirrung zu vermeiden ist es wichtig folgende Unterscheidung hervorzuheben: Wenn von **Symbolen** die Rede ist, so werden in der **Lexikalischen Analyse**, der **Syntaktischen Analyse** und der **Code Generierung**, auf diesen verschiedenen Ebenen unterschiedliche Konzepte als Symbole bezeichnet.

In der Lexikalischen Analyse sind einzelne **Zeichen eines Zeichensatzes** die Symbole.

In der Syntaktischen Analyse sind die **Tokennamen** die Symbole.

In der Code Generierung sind die **Bezeichner von Variablen, Konstanten und Funktionen** die Symbole^a.

^aDas ist der Grund, warum die Tabelle, in der Informationen zu Identifiern gespeichert werden aus Kapitel 3 Symboltabelle genannt wird.

Definition 2.16: Literal

Eine von möglicherweise vielen weiteren **Darstellungsformen** für ein und denselben **Wert**.

Eine weitere Aufgabe der **Lexikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen `␣`, Newline `\n`² und Tabs `\t` aus dem Inputstring herauszufiltern, entweder vom Lexer oder schon bevor der Inputstring an den Lexer übergeben wird. Nur das, was für die **Syntaktische Analyse** wichtig ist, soll weiterverarbeitet werden, alles andere wird herausgefiltert.

¹Die Programmiersprache Python erlaubt es z.B. diesen Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen

²In Unix Systemen wird für Newline das ASCII Symbol **line feed**, in Windows hingegen die ASCII Symbole **carriage return** und **line feed** nacheinander verwendet. Das wird aber meist durch die verwendete Programmiersprache, die man zur Implementierung des Lexers nutzt wegabstrahiert.

Die **Grammatik** G_{Lex} , die zur Beschreibung der Token T einer regulären Sprache L_{Lex} verwendet wird, ist üblicherweise **regulär**, da ein typischer **Lexer** immer nur **ein oder wenige Symbole** vorausschaut^a, unabhängig davon, was für Symbole davor aufgetaucht sind. Die übliche Implementierung eines **Lexers** merkt sich nicht, was für Symbole davor aufgetaucht sind, der **Kontext** in dem ein Symbol auftaucht ist also **nicht wichtig**.

^aMan nennt das auch einem **Lookahead** von 1 oder k

2.5 Syntaktische Analyse

Die vom **Lexer** identifizierten **Token** der **Sprache** L_{Lex} werden nun im Folgenden

Der **Parser** nutzt **Token** T als Wegweiser, um herauszufinden,

Definition 2.17: Parser

*Ein Programm, dass eine **Eingabe** in ein für die **Weiterverarbeitung** taugliche Form bringt.*

An dieser Stelle könnte möglicherweise eine Begriffsverwirrung entstehen.

In Bezug auf einen Compiler hat ein Parser meist die Aufgabe aus einem **Inputstring** einen **Derivation Tree** zu generieren.

Definition 2.18: Recognizer

Definition 2.19: Konkrete Syntax

Definition 2.20: Derivation Tree

Definition 2.21: Abstrakte Syntax

Definition 2.22: Abstrakte Syntax Tree

Definition 2.23: Transformer

Definition 2.24: Visitor**2.6 Code Generierung****Definition 2.25: Pass****2.7 Fehlermeldungen****Definition 2.26: Fehlermeldung**

3 Implementierung

3.1 PicoC und RETI

ASTNode

3.2 Grammatiken

3.2.1 Umstzung von Präzidenz

Die PicoC Sprache hat dieselben Präzidenzregeln implementiert, wie die Sprache C¹. Die Präzidenzregeln von PicoC sind in Tabelle 3.2.1 aufgelistet.

Präzidenz	Operator	Beschreibung	Assoziativität
1	a() a[] a.b	Funktionsaufruf Indezzugriff Attributzugriff	Links, dann rechts →
2	-a !a ~a *a &a	Unäres Minus Logisches NOT und Bitweise NOT Dereferenz und Referenz, auch Adresse-von	Rechts, dann links ←
3	a*b a/b a%b	Multiplikation, Division und Modulo	Links, dann rechts →
4	a+b a-b	Addition und Subtraktion	
5	a<b a<=b a>b a>=b	Kleiner, Kleiner Gleich, Größer, Größer gleich	
6	a==b a!=b	Gleichheit und Ungleichheit	
7	a&b	Bitweise UND	
8	a^b	Bitweise XOR (exclusive or)	
9	a b	Bitweise ODER (inclusive or)	
10	a&&b	Logisches UND	
11	a b	Logisches ODER	
12	a=b	Zuweisung	Rechts, dann links ←
13	a,b	Komma	Links, dann rechts →

Tabelle 3.1: Präzidenzregeln von PicoC

¹C *Operator Precedence* - cppreference.com.

3.3 Lexikalische Analyse

3.3.1 Verwendung von Lark

3.3.2 Basic Parser

3.4 Syntaktische Analyse

3.4.1 Verwendung von Lark

3.4.2 Derivation Tree

3.4.3 Derivation Tree Vereinfachung

3.4.4 Early Parser

3.5 Code Generierung

3.5.1 Passes

PicoC-Shrink Pass

PicoC-Blocks Pass

PicoC-Mon Pass

Definition 3.1: Symboltabelle

RETI-Blocks Pass

RETI-Patch Pass

RETI Pass

3.5.2 Umsetzung von Pointern und Arrays

3.5.3 Umsetzung von Structs

3.5.4 Umsetzung von Funktionen

3.5.5 Umsetzung kleinerer Details

3.6 Fehlermeldungen

3.6.1 Error Handler

4 Ergebnisse und Ausblick

4.1 Funktionsumfang

4.2 Qualitätskontrolle

4.3 Kommentierter Kompiliervorgang

4.4 Erweiterungsideen

A Appendix

A.1 Konkrete und Abstrakte Syntax

A.2 Bedienungsanleitungen

A.2.1 PicoC-Compiler

A.2.2 Showmode

A.2.3 Entwicklertools

Literatur

Online

- *C Operator Precedence* - *cppreference.com*. URL: https://en.cppreference.com/w/c/language/operator_precedence (besucht am 27.04.2022).
- *lecture-notes-2021*. 20. Jan. 2022. URL: <https://github.com/Compiler-Construction-Uni-Freiburg/lecture-notes-2021/blob/56300e6649e32f0594bbbd046a2e19351c57dd0c/material/lexical-analysis.pdf> (besucht am 28.04.2022).
- *What is the difference between a token and a lexeme?* NewbeDEV. URL: <http://newbedev.com/what-is-the-difference-between-a-token-and-a-lexeme> (besucht am 17.06.2022).