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

Betreung:
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

Abbildungsverzeichnis

Tabellenverzeichnis

Definitionen

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: Chromsky 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]

[a] Als Beschreibungswerkzeug können aber auch z.B. reguläre Ausdrücke hergenommen werden.
[b] *What is the difference between a token and a lexeme?*

Diese Folgen endlicher Symoble 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]

[a] *What is the difference between a token and a lexeme?*

Diese **Lexeme** werden vom **Lexer** im **Inputstring** identifziert 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 $lex : \Sigma^* \rightharpoonup (N \times W)^*$, 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]

[a] *lecture-notes-2021*.

Ein **Lexer** ist im Allgemeinen eine **partielle Funktion**, da es Zeichenfolgen geben kann, die kein **Pattern** eines **Tokens** der Sprache L_{Lex} matchen. In Bezug auf eine Implementierung, wird, wenn der Lexer Teil der Implementierung eines Compilers ist, in diesem Fall eine **Fehlermeldung** ausgegeben.

Falls der Inputstring, den der **Lexer** erhält noch Leerzeichen ␣ oder andere für die **Syntaktische Analyse** unwichtige Zeichen enthält ist das auch im Sinne der Definition. Dann wird diesen das **leere Wort** ϵ zugeordnet, denn $\epsilon \in \Sigma^*$.

In den G_{Lex} Grammatiken einiger Programmiersprachen sind allerdinds alle möglichen Zeichenfolgen allein dadurch schon möglich, weil diese Programmiesprachen das Konzept eines **Identifiers** o.ä. umsetzen, der alle möglichen Zeichenfolgen abfängt[1]. Wodurch der Lexer wiederum doch eine **linkstotale partielle Funktion** ist, die man im Allgemeinen einfach als Funktion bezeichnet: $lex : \Sigma^* \to (N \times W)^*$.

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 darstellt werden, einmal als ASCII-Zeichen `'c'` und des Weiteren auch in Dezimalschreibweise als `99`[2]. Der **Tokenwert** ist jedoch der letztendliche Wert an sich, unabhängig von der Darstellungsform.

Um Verwirrung verzubäugen ist es wichtig folgende Unterscheidung hervorzuheben: Wenn von **Symbolen** die Rede ist, so werden in der **Lexikalischen Analyse**, der **Syntaktische 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 Funktionnen** die Symbole[a].

[a] Das ist der Grund, warum die Tabelle, in der Informationen zu Identifiern gespeichert werden aus Kapitel 3 Symboltabelle genannt wird.

[1] Bei der Grammatik von C und auch PicoC ist das allerdings nicht der Fall, weil Identifier dort nicht mit einer Zahl anfangen dürfen.

[2] Die Programmiersprache Python erlaubt es z.B. diesern Wert auch mit den Literalen `0b1100011` und `0x63` darzustellen.

Definition 2.16: Literal

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

Eine weitere Aufgabe der **Lekikalischen Analyse** ist es jegliche für die Weiterverarbeitung unwichtigen Symbole, wie Leerzeichen ␣, Newline `\n`[3] 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.

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**.

[a] Man 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 **Weiterverbeitung** taugliche Form bringt.*

An dieser Stelle könnte möglicherweise eine Begriffsverwirrung enstehen.

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: Konkrette Syntax

Definition 2.20: Derivation Tree

[3] In 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 Porgrammiersprache, die man zur Inplementierung des Lexers nutzt wegabstrahiert.

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 Lexikalische Analyse

3.1.1 Verwendung von Lark

3.1.2 Basic Parser

3.2 Syntaktische Analyse

3.2.1 Verwendung von Lark

3.2.2 Umsetzung von Präzidenz

Die PicoC Sprache hat dieselben Präzidenzregeln implementiert, wie die Sprache C[1]. Die Präzidenzregeln von PicoC sind in Tabelle 3.2.2 aufgelistet.

Präzidenz	Operator	Beschreibung	Assoziativität

1	`a()`
	`a[]`
	`a.b`
		Funktionsaufruf
		Indexzugriff
		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`	Logiches 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

[1] *C Operator Precedence - cppreference.com*.

3.2.3 Derivation Tree Generierung

3.2.4 Early Parser

3.2.5 Derivation Tree Vereinfachung

3.2.6 Abstrakt Syntax Tree Generierung

ASTNode

PicoC Nodes

RETI Nodes

3.3 Code Generierung

3.3.1 Passes

PicoC-Shrink Pass

PicoC-Blocks Pass

PicoC-Mon Pass

Definition 3.1: Symboltabelle

RETI-Blocks Pass

RETI-Patch Pass

RETI Pass

3.3.2 Umsetzung von Pointern und Arrays

3.3.3 Umsetzung von Structs

3.3.4 Umsetzung von Funktionen

3.3.5 Umsetzung kleinerer Details

3.4 Fehlermeldungen

3.4.1 Error Handler

4 Ergebnisse und Ausblick

4.1 Funktionsumfang

4.2 Qualitätskontrolle

4.3 Kommentierter Kompiliervorgang

4.4 Erweiterungsideen

A Appendix

A.1 Konkrette 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).